MT-WAVE: Profiling Multi-Tier Web Applications

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Anthony Arkles

PERMISSION TO USE

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
University of Saskatchewan
Saskatoon, Saskatchewan S7N 5A9

ABSTRACT

The web is evolving: what was once primarily used for sharing static content has now evolved into a platform for rich client-side applications. These applications do not run exclusively on the client; while the client is responsible for presentation and some processing, there is a significant amount of processing and persistence that happens server-side. This has advantages and disadvantages. The biggest advantage is that the user's data is accessible from anywhere. It doesn't matter which device you sign into a web application from, everything you've been working on is instantly accessible. The largest disadvantage is that large numbers of servers are required to support a growing user base; unlike traditional client applications, an organization making a web application needs to provision compute and storage resources for each expected user. This infrastructure is designed in tiers that are responsible for different aspects of the application, and these tiers may not even be run by the same organization.

As these systems grow in complexity, it becomes progressively more challenging to identify and solve performance problems. While there are many measures of software system performance, web application users only care about response latency. This "fingertip-to-eyeball performance" is the only metric that users directly perceive: when a button is clicked in a web application, how long does it take for the desired action to complete?

MT-WAVE is a system for solving fingertip-to-eyeball performance problems in web applications. The system is designed for doing multi-tier tracing: each piece of the application is instrumented, execution traces are collected, and the system merges these traces into a single coherent snapshot of system latency at every tier. To ensure that user-perceived latency is accurately captured, the tracing begins in the web browser. The application developer then uses the MT-WAVE Visualization System to explore the execution traces to first identify which system is causing the largest amount of latency, and then zooms in on the specific function calls in that tier to find optimization candidates. After fixing an identified problem, the system is used to verify that the changes had the intended effect.

This optimization methodology and toolset is explained through a series of case studies that identify and solve performance problems in open-source and commercial applications. These case studies demonstrate both the utility of the MT-WAVE system and the unintuitive nature of system optimization.

ACKNOWLEDGEMENTS

Thank you very much to everyone who helped make this work possible: Dr. Dwight Makaroff, my incredibly patient supervisor; Dr. Derek Eager and Dr. Carl Gutwin, my perpetually tolerant committee; my parents, who encouraged me to embark on this journey; and Karen Gwillim, my infinitely supportive partner.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

List of Abbreviations

AJAX	Asynchronous Javascript and XML
APC	Alternative PHP Cache
API	Application Programming Interface
AST	Abstract Syntax Tree
BSD	Berkeley Software Distribution
CMS	Content Management System
CSS	Cascading Stylesheet
DAG	Directed Acyclic Graph
DNS	Domain Name System
DOM	Document Object Model
DTD	Document Type Definition
GPS	Global Positioning System
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IIS	Microsoft Internet Information Services
JSON	Javascript Object Notation
LALR	Look-ahead LR (bottom-up) Parser
MSIL	Microsoft Immediate Language
PMU	Performance Monitoring Unit
RPC	Remote Procedure Call
RUM	Real User Monitoring
SGML	Standard Generalized Markup Language
SQL	Structured Query Language
TCP	Transmission Control Protocol
UDP	Unreliable Datagrame Protocol
USE	Utilization, Saturation, and Errors
URL	Uniform Resource Locator
VM	Virtual Machine
XML	Extensible Markup Language

CHAPTER 1

INTRODUCTION

1.1 Context

1.1.1 Fingertip-to-Eyeball Performance

One of the key concerns while evaluating web application performance is responsiveness: how long does it take for an action to complete? From a user's point of view, the entire system is treated as a black box. They don't care about database transaction throughput, or load balancers, or any other technical aspect; they simply click on something in the application and hope that their request completes quickly.

Since this so-called "fingertip-to-eyeball performance" [23] is the primary way that users experience web application performance, it is the primary metric in this research. Tools and techniques that enable a developer to determine the root cause of these performance problems concerning this metric is the topic of this thesis.

Though the focus is on solving a high-level problem, that cannot be accomplished without looking at the details of how the application executes. To solve the fingertip-to-eyeball problem, it is necessary to look at the problem from varying levels of detail: start from the measured user-perceived latency and dig into the system performance tier-by-tier to determine which specific piece of the application requires optimization.

Existing tools can effectively solve pieces of the problem, but do not solve the whole problem. Web browsers can provide a simple report showing how long each request takes. Profilers can show which functions take the longest time to compute on the server. Databases allow a developer to identify slow queries. But none of these tools can answer the fundamental question: "When I click on this button in my application, why does it take so long for the result to show up on my screen?"

1.1.2 Web Applications & Performance

The primary focus of this research is Web 2.0 applications – rich applications with a high degree of user interactivity. These differ from traditional websites in that they are not made primarily of static content meant solely for consumption, but rather have a significant amount of user-generated and user-specific content.

With the introduction of smartphones, web applications started to grow as a straightforward way to have the same application on multiple platforms while maintaining only one codebase; the same application would work on a desktop, tablet, or smartphone with minimal modification [9]. Web browser functionality continued to grow as well; what started

out as a simple platform for viewing documents continually gains more and more functionality. Canvas and WebGL APIs provide web applications with the ability to render dynamic 2D and 3D scenes, the Web Audio API[1] provides direct access to the sound card, and technologies like PhoneGap allow applications to interact directly with many more operating system features.

A significant benefit of this system is that deployment and updating of the application is much simpler; instead of being installed directly on the target device, most of the application is downloaded as needed. To update the application, it only needs to be updated on the server and all clients will automatically use the new version. While this approach to application development has many advantages, there are trade-offs.

Since web applications require an Internet connection to function, there is potential for significant latency problems due to connectivity and capacity limitations. Initiating an action within the application will frequently require a connection to be made to the server before the results are available. Once the connection is established, a request is sent, the server processes the data (often by forwarding the request to other servers), and the response is sent back to the client. When there's a performance problem in a web application, it can be significantly more challenging to debug than in a traditional native application. Profilers are used to determine the source of latency in an application running locally on a device, but in a web application, a profiler will simply inform the developer that the latency is due to a network connection. To further complicate the issue, servers can only measure processing latency; when there's latency caused by a high-latency connection between the client and the server, tools on the server can only see that the request was processed quickly; there's no way for the server to tell that the response took a long time to get back to the client.

In practice, user studies have shown some basic thresholds for user-perceived performance [42] and studies at Google and Amazon have shown that small changes in performance can result in significant changes in browsing and purchasing behaviour. Greg Linden from Amazon reported[2] a 1% loss in sales due to an additional 100 ms of latency, and Marissa Mayer from Google reports[3] a 1% loss in user searches due to an additional 500*ms* of latency (Google sells advertisements on search results, so each lost search is a lost advertisement sale). Many popular "application categories" have several competitors (e.g. GitHub vs. BitBucket, Flickr vs. 500px, etc.), and user-perceived performance can factor into purchasing decisions when evaluating competing applications. Eric Shurman (Bing) and Jake Brutlag (Google) have also presented results [53] showing a 4% loss in revenue caused by 2000*ms* of added latency.

1.2 Motivation

The web has been going through a dramatic shift in the last several years; the web started as a fat-server, thin-client system, but much of this processing is being rapidly shifted to the client. As this shift has been taking place, the tools developers use to manage software performance have not kept up. When the majority of computation was limited

[1] https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API

[2] At a Data Mining presentation at Stanford, 2006-11-29, http://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnxnbGluZGVufGd4OjVmZDIzMWMzMGI4MDc3OWM

[3] At the O'Reilly Velocity 09 conference keynote, 2009-06-24, http://velocityconf.com/velocity2009/public/schedule/detail/8913

to a single tier (the server), one could gain significant performance tuning insight by monitoring that system; as web applications shift to a more distributed system, multi-tier monitoring is necessary to understand the whole-system performance.

This research was strongly motivated by industrial experience; local organizations were struggling to identify the source of performance problems in their applications. Google App Engine was gaining in popularity, and for many developers this was their first experience using a distributed architecture for web applications. The iPhone was new, and mobile web applications were growing in popularity. Working with developers, they were capable of building applications for these platforms but struggled to identify and fix performance problems. Research was done to try to find suitable commercial or open-source tooling, but all of the available solutions were insufficient.

Shortly after this research began, it became clear that large companies had developed internal tools to begin to address these problems (e.g. Google Dapper [54]), but these were large internal projects that were not made publicly available. Smaller organizations simply did not have the resources available to develop tools at this scale, nor was it a market that they intended to enter. These were businesses focused on building applications for consumers, and they were not interested in building a suite of tracing applications for developers.

As a result, development continued to be impeded by the lack of tooling and methodology. Developers would try to solve problems using intuition and guesswork, occasionally in a systematic way but frequently with very little planning or guidance. The existing tools would provide hints about where the problem may be, but (as demonstrated in Section 4.3.1) frequently a proposed improvement has unintended effects that further degrade system performance instead of improving it.

The set of tools, techniques, and methodology in this thesis is the result of looking carefully at the struggles experienced industrially as developers transitioned from static web pages served on single servers to fully interactive web applications served from distributed systems. It is the synthesis of a large body of existing academic research, combined and targeted towards solving industrial problems.

1.3 Thesis Statement

This thesis investigates the feasibility and utility of multi-tier tracing techniques applied to modern web applications. There exists a large body of research focused on distributed systems, large-scale system tracing, and methodologies for finding the root cause of latency problems, but this body of research has not been synthesized into a set of tools and techniques for multi-tier web applications. This work documents an effective approach to low-overhead multi-tier web application tracing.

1.4 Contributions

This thesis investigates the use of multi-tier tracing in the context of web applications. There are two main contributions:

- A tracing methodology and implementation suitable for doing multi-tier tracing in web applications;
- A visualization system for interpreting execution traces in web applications.

1.5 Outline

The remainder of this thesis document is organized as follows. Chapter 2 provides an outline of the technologies and protocols used in modern web applications, followed by a summary of the difficulties encountered when attempting to trace execution flow though a distributed system. It continues with a description of different strategies for solving performance problems, and finishes with a survey of existing tracing, profiling, and analysis techniques.

Chapter 3 summarizes the specific techniques used in this thesis for tracing, web browser event capture, and instrumentation. The event model and X-Trace extensions to facilitate tracing in web applications follow. The specific programming language tracing modules are explained, along with a brief tour of the visualization system. Finally, there is a description of techniques that can be used with this system for making scripted, repeatable experiments and data archival.

Chapter 4 contains the case studies: first, an evaluation of the overhead imposed by the MT-WAVE tracing system; second, performance enhancements that were applied to three different web applications; and finally, a description of how MT-WAVE was used to find a bug in MT-WAVE.

Chapter 5 summarizes the findings and explains potential future work that can come out of this work; the web is constantly evolving, and the performance evaluation tools that we use will have to continue to adapt to the ever-changing ecosystem.

CHAPTER 2

BACKGROUND & RELATED WORK

2.1 Introduction

This chapter describes the implementation of modern web applications, starting with the technologies used and the distributed network structure. This leads to a discussion of the more theoretical aspects of distributed events and tracing, including various high-level strategies for doing performance analysis in complex systems. Finally, there is a large survey of specific techniques and implementations developed academically and industrially to do performance measurement and analysis.

2.2 Technologies and Architectures for Web Applications

Modern web applications are designed and built using a diverse set of technologies and architectures. This section outlines examples from each tier of the web application stack and explains how the different pieces fit together. An exhaustive catalogue of technologies and architectures would be prohibitively large; as such, this is a collection of archetypical technologies that provide real-world examples of the varied approaches taken by different architectures and frameworks.

2.2.1 Technologies

To illustrate why multi-tier tracing is necessary for web application performance monitoring, it is important to understand how many different technologies are used in the different tiers of the application stack and how they interact. At each tier, there is a spectrum of complexity; some technologies provide very little beyond the bare minimum required to function, while others provide a significant framework around which to build an application. Any one of these components could be the root cause of a performance problem; determining the source of and solving performance problems requires evaluating performance at each tier of the stack.

2.2.1.1 Protocols

In the ecosystem of web application technologies, the underlying protocol HTTP is the only standardized component. There are a diverse set of clients, servers, backend and frontend technologies, but they all speak HTTP.

HTTP is a "stateless" client-server protocol with a request/response model. Clients make requests for resources and servers reply with responses; in the standard version, there is no way for a server to send data to a client without the client first making a request. While the base protocol is stateless, there are extension used to emulate statefulness, e.g. Cookies [4]. On its own, HTTP is content-agnostic; it simply acts to encapsulate the request/response model and provide metadata to describe the data (via HTTP headers).

In general, HTTP can be used to transmit any type of digital content. Frequently, it is simply used as a way to download files from an HTTP server onto a client machine (e.g. static content like images, HTML documents, movies, etc). There are, however, several data formats that are frequently used in web applications that are directly parsed and utilized by HTTP clients.

The most well-known application-specific format is the Hypertext Markup Language (HTML). This format has been part of "the web" since the beginning and has evolved over time; HTML5 [24] is the most recent version of HTML, and it has been widely (although not completely) adopted by modern web clients. HTML is a structured format that, while quite expressive for describing documents, has complex parser requirements. The parser is generally quite liberal in attempting to parse non-conforming documents. HTML is a very effective format for transferring content that will be rendered for direct user consumption, but due to the significant parsing overhead required and potential ambiguity in non-conforming documents, it is somewhat inappropriate as a data interchange format.

The eXtensible Markup Language (XML) [6] is a common data interchange format similar to HTML—both have roots in Standard Generalized Markup Language (SGML). XML has much stricter parsing and correctness requirements than HTML; a non-conforming XML document is considered an error and will be rejected by the parser. XML is extensible through Document Type Definitions (DTDs). The XML specification describes the format of the document, and the DTD describes the structure of the data in the document. XML, like HTML, has a significant parsing overhead.

JavaScript Object Notation (JSON) [26] is a simpler structured data format that is "easy for humans to read and write" and "easy for machines to parse and generate." While this format is missing a number of features available in XML (e.g. there is no built-in mechanism for verifying that the data structures conform to any particular specification), it has gained wide adoption in web applications due to its simplicity. JSON parsing libraries are quite simple and are readily available.

2.2.1.2 Languages

There are a number of different programming languages used for web applications. This subsection is not meant to provide an exhaustive list, but rather a sampling of popular languages. It is very difficult to gauge the absolute popularity and prevalence of these different languages, since most sites do not advertise what languages are being used behind-the-scenes; in this survey, there are subjective interpretations of the current state-of-the-industry that have not been studied rigorously but are rather the result of years of industrial experience.

Python [34] is an interpreted language that was not explicitly designed for web programming. It is fundamentally a procedural language, with support for both object-oriented programming and functional programming. The standard

libraries contain several modules to aid in the development of web applications (e.g. cgi and BaseHTTPServer). Ruby [57] is similar to Python, with different syntax but similar underlying principles: an interpreted procedural language with strong support for object-oriented and functional programming paradigms.

Perl [11] was one of the first common web application programming languages, although it was not designed specifically for this task. In the '90s, Perl was one of the main languages that powered the web, although it has (subjectively) fallen out of favour for newer languages. PHP [55] gained popularity in the late 90s, but unlike Perl, has continued to hold a dominant position in the web application ecosystem. While widely criticized for many of its design choices, its popularity remains high primarily due to ease of use and hosting momentum. It is incredibly easy to find a low-cost hosting provider that will run your PHP site, and the "Hello World" of a PHP web application is simply one file with one line of code.

Java is different from the above languages; it is a purely object-oriented language that is pre-compiled and run in a virtual machine. Java has been relatively popular in the enterprise space since its introduction; originally developed by Sun Microsystems, now owned by Oracle, Java was designed to appeal to large corporations with large, complex codebases. It fills this role well, but at the cost of code, deployment, and maintenance complexity. In contrast to PHP, a Java web application "Hello World" is non-trivial amount of code that requires a specialized web server to run.

Go [15] is a relative newcomer to the programming language arena. First released by Google in 2009, Go was developed to address the problem that "No major systems language has emerged in over a decade, but over that time the computing landscape has changed tremendously."[1] It is a "compiled, concurrent, garbage-collected, statically-typed language"[2] that was designed at Google to specifically address the problems they were having with existing programming languages at their operational scale: slow builds, uncontrolled dependencies, difficulty of writing automatic tools, and duplication of effort. For example, the Go language and build system is designed with explicit support for version control and modularity. Instead of either requiring a dependent library to be installed on the system or requiring the import of that library's source code into a project, the build system is designed to build external dependencies on demand; the developer need only add the URL to the source code repository in the project configuration. It's not a particularly popular language yet (the others have had 15-20 years more than Go to gain popularity), but it is definitely worth keeping an eye on in the future.

C++ is a veteran object-oriented, compiled, statically-typed programming language. While it has never been particularly popular as a web programming language, many of the high performance services that web applications rely on are written in C++. To compensate for the performance lost due to interpreter overhead in typical interpreted web application languages, specific performance-critical modules for PHP, Python, and Ruby are often written in either C or C++; this means that the high-level language code describes the structure of the computation, and compiled C/C++ code performs the actual computations.

Erlang is a concurrent, virtual-machine based, functional programming language designed by Ericsson. Initially designed as a proprietary language for high-reliability telecom applications, it was made open source in the late 90s.

[1] http://golang.org/doc/faq
[2] http://talks.golang.org/2012/splash.article

Like C++, it hasn't gained a lot of popularity for writing web applications, but many backend services use Erlang for high availability and reliability. It also has a reputation for performance, reliability, scalability, and engineering productivity. In a recent conference talk [46], Rick Reed (engineer at WhatsApp) talked about the scale of their Erlang-based system: 19 billion messages/day input, 40 billion messages/day output, and 147 million concurrent connections.

Javascript is a dynamic interpreted language that is built into almost every web browser on the Internet. In 2009, NodeJS [7] was released, bringing Javascript from being a solely client-side language to being a language that can run on both the client and the server. NodeJS is unique in that, contrast to most programming languages, calls to the standard library are almost universally non-blocking and instead are passed callback functions that execute once the operation has completed.

2.2.1.3 Web Server

Web servers for serving web applications vary in complexity. Apache has been the most popular web server for a long time,[3] although there are several other web servers that have been gaining in popularity; in particular, nginx [41] has risen to the #3 spot in the 2014 Netcraft survey and takes a very different approach to serving web applications than Apache does.

Apache has, over time, gained support for different web application platforms; for example, mod_php serves the output from PHP scripts, mod_wsgi serves the output from Python scripts, and Phusion Passenger supports Ruby. These modules embed the script interpreter directly into the web server process. The Apache worker model forks a number of processes that wait for an incoming request, determine how to handle it (possibly by loading and executing a script), and return the result.

In the mod_php case, PHP scripts are loaded from scratch every time a request comes in. The script is read from disk, parsed, and the result of execution is returned to the client. Since the script is read from disk for every request, this process can be slow. There are a few workarounds for this (e.g. bytecode caching through APC,[4] but the script is still loaded into the PHP interpreter each time a request is received.

Phusion Passenger and mod_wsgi take an alternative approach: the worker will load and execute the application initialization well before a request is received; once a request is received, the system is already prepared to process it. This is an improvement over the mod_php approach, but still incurs a significant amount of repeated initialization.

The nginx model is significantly different; while Apache could be called "full featured," nginx would be more appropriately called "minimalist." In the nginx model, the web application is started externally and runs perpetually (contrast to being reloaded and initialized for each request); communication between nginx and the application happens over TCP sockets or Unix domain sockets (preferable, for performance reasons).

[3] http://news.netcraft.com/archives/2014/04/02/april-2014-web-server-survey.html
[4] http://php.net/manual/en/book.apc.php

2.2.1.4 Web Clients

Web clients also vary in complexity, from the lowest-level APIs built into the standard libraries for programming languages, to command-line tools like `curl` and `wget`, to full-featured web browsers like Chrome, Firefox, and Internet Explorer.

As an example, Python's urllib library[5] provides a simple interface for creating HTTP connections and receiving the responses. This interface simply retrieves the response and returns it raw to the user; it makes no attempt to parse the response into any kind of data structure or visual representation. It is up to the application to determine the format of the returned data and handle it appropriately.

Many mobile applications use similar libraries to interact with web services. Android has an HttpClient class that allows the application developer to make requests and receive responses; likewise, iOS has the NSURLSession and NSURLConnection classes to support HTTP communication.

Tools like `wget` and `curl` are command-line tools that allow users to retrieve content from HTTP servers and save the content to disk. Like the low-level libraries, they make no attempt to parse the response or perform any action with the response other than saving it to the local machine. Their sole task is to provide a simpler interface for creating HTTP requests than writing code.

Browsers provide the most functionality: they make an HTTP request, parse the response into a standard format (e.g. the HTML Document Object Model), make additional requests for resources referenced in the document, execute included Javascript, and render the DOM into a visual interface for the user. When a user interacts with the rendered page, the browser determines what to do with the event: scroll the current page, load a new web page, or execute a page-defined Javascript callback function. Unlike the other tools, web browsers are long-running tasks that a user will interact with over and over.

A web browser environment has three main pieces of functionality:

DOM The Document Object Model is a tree-based data structure that represents the logical structure of the web page. Nodes are referred to as *DOM Elements*;

CSS Cascading Stylesheets describe the visual representation of the DOM; and

JavaScript provides interactivity to otherwise static content (it can react to browser events and manipulate the DOM and CSS).

The DOM is initially generated by parsing HTML using a rather convoluted algorithm [61]; because the initial implementation of HTML was rather liberal about the parsing process, browsers strive to handle parsing errors gracefully for the sake of backward compatibility (i.e. a new web browser that failed to display old web pages would have a hard time gaining acceptance; even though the web page markup is incorrect, users will complain that "it worked fine in my old browser").

[5] `https://docs.python.org/3/howto/urllib2.html`

CSS is parsed in a much more straightforward manner; the grammar of CSS can be parsed with a simple LALR(1) parser [5]. The CSS completely defines the visual appearance of the DOM Elements; browsers include a set of default styles for common elements, but pages using the default styles are quite bland. Javascript, like HTML, has a complex parsing algorithm to deal with backwards compatibility [25].

2.2.1.5 Backend Application Frameworks

Choosing which level of framework complexity is required for a given project is primarily an exercise in weighing various engineering trade-offs. Larger frameworks include a lot of functionality that may not be required and can affect performance, but smaller frameworks may be lacking features required for a given project.

The simplest is bare PHP, where there is no framework used at all; the web server maps a URL to a script and the output of the script is returned verbatim as the response. In languages like Python or Ruby, which were not designed specifically as web languages, a significant amount of generic code is required to provide similar bare functionality (typically copy and pasted from the documentation). Independent of the framework, a common architecture and set of processing steps has evolved and solidified:

- Parse request;
- Pass the request through Request Middleware;
- Execute the application's handler for the request (sometimes called a Controller);
- Generate markup for the handler's response (sometimes called a View, sometimes called a Template);
- Pass the response through Response Middleware; and
- Return the response to the web server.

Flask [22] (Python) and Sinatra [62] can be considered "minimalist" frameworks. These handle the basic request/response model and mapping URLs to specific functions or objects to call, but very little else.

More complete frameworks like CodeIgniter [21] (PHP), Django [14] (Python), or Ruby On Rails [48] provide much more functionality, including HTML templating and form handling, database abstraction and format migration, multiple response output formats, and many more features.

2.2.1.6 Frontend Application Frameworks

Javascript is the main language that browsers support for front-end applications, and is the only programming language that has had consistent browser support. Initially used for simple interactivity on web pages, its use has expanded far beyond the initial intention. The XMLHTTPRequest library, [59] for example, provides a way for Javascript to make HTTP requests without requiring a page to be reloaded.

Javascript in the web browser is generally event-driven. It does not run continuously; long-running Javascript code will lock the browser entirely. This programming model is a significant departure from the back-end programming model, where the code executes to completion on every request.

Like the backend frameworks, there are a variety of frontend frameworks that provide different levels of functionality. Like PHP, JavaScript does not strictly require a framework, but frameworks can provide well-defined abstractions and a library of functionality instead of "reinventing the wheel" on every new project. There are many different frameworks available; the following is a survey of concrete examples of the different levels of abstraction, structure, and complexity that different frameworks offer.

An example of a relatively simple framework is jQuery [47]; it provides an API to simplify the HTML DOM API (and associated cross-browser incompatibilities) and provides convenience methods for event handling and making additional HTTP requests.

Foundation[6] and Bootstrap[7] provide more functionality than jQuery, but provide no specific application structure. They both come with a large collection of reusable markup and rendering styles to enable rapid application development, and large collections of reusable "widgets" that can be used in an application (e.g. tab and menu bars, icons, drop-downs, navigation).

Backbone.js [43] provides more structure to web applications by providing a structure for data models and collections, events, views, and routing between different views. This is still a relatively low-level framework; it provides tools, but it is up to the application developer to design the specific structure that they would like for their application. Frameworks like Backbone are typically considered "mix and match"; you are free to use whatever pieces of the framework you'd like without being required to fit your entire application into a rigid framework-defined structure.

Ember.js [13] provides a more rigid framework for developing applications. Code is explicitly broken into models, views/templates, and controllers to fit into the provided structure. The Router class is designed to specifically work with Ember models and views, and requires you to use their defined models to get the desired functionality.

Angular [51] is one of the most rigid and complete frameworks. Not only does it provide a model-view-controller-style framework, but it also extends HTML itself to provide a specific model for application development. Their documentation explicitly states that "Angular simplifies application development by presenting a higher level of abstraction to the developer. Like any abstraction, it comes at a cost of flexibility. In other words not every app is a good fit for Angular."

2.2.2 Web Application Architecture

The first generation of content served on the web was generally static HTML. A web browser would request a specific URL, the web server would return a response, and the browser would render it. When a user clicked on a link in the page, a new URL was requested and the process would start all over again. Figure 2.1 shows the typical client-server interactions in this model.

[6] http://foundation.zurb.com/
[7] http://getbootstrap.com/

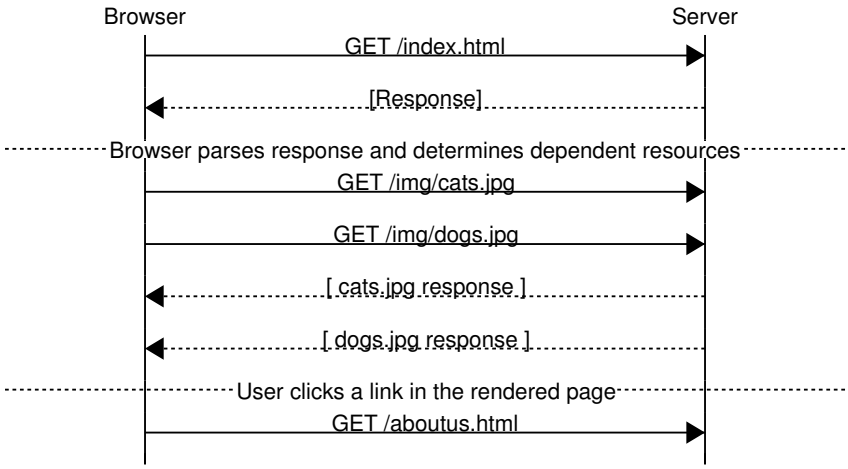


Figure 2.1: Typical client-server interaction in traditional website architecture.

As the need for more interactive content arose, the traditional web application emerged. This is built from just a few pieces:

- Web server (e.g. Apache, IIS);
- Application language (e.g. Python, PHP, Perl); and
- Web browser.

In this architecture, the web browser makes an HTTP request for an HTML document to the web server, which forwards the request to the application. The application returns the HTML document which is then rendered in the web browser.

The document frequently consists of other resources, which are also requested (images, stylesheets, etc). When a user interface element is clicked, a new HTTP request is made for a new HTML document and the process starts over.

As client-side interactivity is added, the model gets a little more complex:

- Web server;
- Back-end application (same as above);
- Front-end application (usually written in JavaScript); and
- Web browser.

In this system, the process begins in the same way as a static website: the user types in a URL, the web browser makes an HTTP request, and then additional resources are loaded based on the returned HTML. This is where the similarities end.

The key difference is what happens when a user interacts with the page. Instead of simply requesting a new URL and starting the page load process over again, the JavaScript front-end application takes over. This code may simply

display a different screen (without loading a new page), it may request some other server-side resource, or it may do some client-side processing. An example of this is shown in Figure 2.2

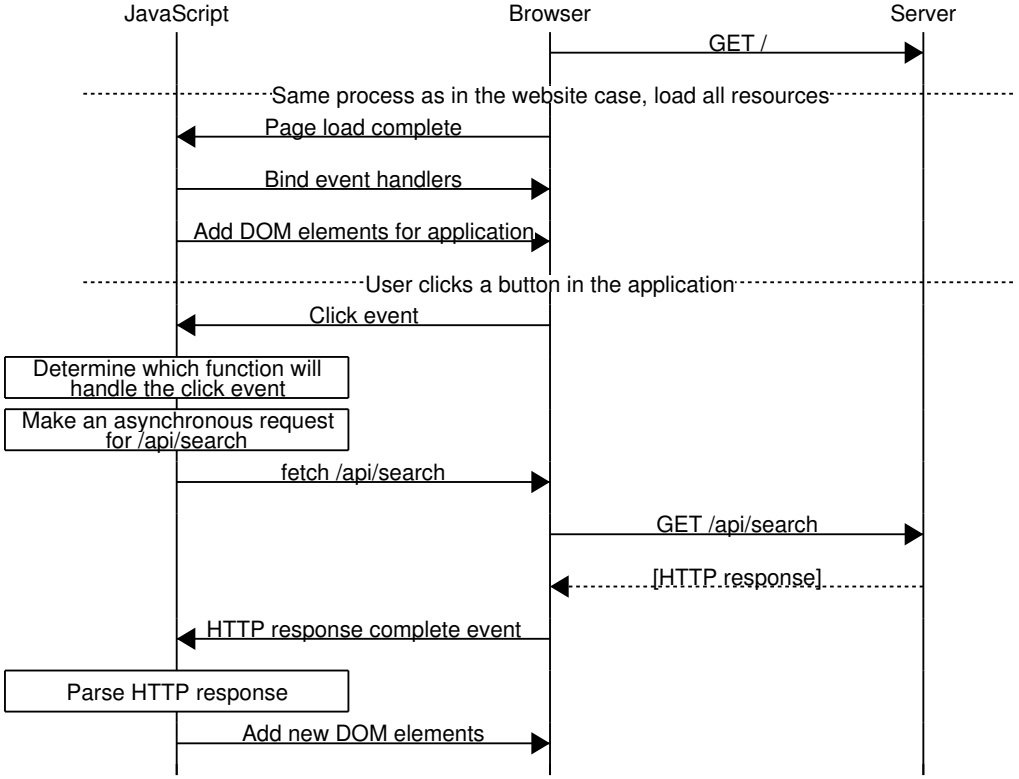


Figure 2.2: Example JavaScript-Browser-Server interaction in a web application.

2.3 Distributed Web Application Architecture

The model described above for web applications fails to account for another important aspect of Web 2.0 applications: scalability. Many of these sites run at a scale that is impossible to serve using a single server; a very common pattern is "horizontal scalability," where a large number of servers are replicated to handle load balancing.

Jayasinghe *et al.* [27] illustrate a distributed version of the RUBBoS[8] benchmark. In this framework, there are 4 tiers: Web Tier, Application Tier, SQL Nodes, and Data Nodes. In this model, the Web Tier serves to route requests to the different nodes in the Application Tier; the Application Tier distributes database requests to the SQL Nodes; and the SQL Nodes use the Data Nodes as a storage backend.

Google App Engine,[9] a cloud computing platform for hosting web applications, uses a similar architecture. The application itself runs in a sandboxed virtual machine (Python, Java, PHP, or Go), and uses a number of different

[8] http://jmob.ow2.org/rubbos.html
[9] https://developers.google.com/appengine/features/

services to provide functionality: Datastore, a schemaless database; Memcache, a distributed in-memory caching system; Images, to manipulate and combine images; MapReduce, for performing computations on large datasets; and many others.

There are many different policies in use for handling this kind of scaling. In some cases, all of the servers are capable of handling every type of request; in other cases, specific clusters of servers are designed to handle specific types of requests. For example, there may be separate clusters of web servers for handling static resources (images, CSS) and for handling dynamic content from application servers. For databases, "primary-secondary" replication[10] requires all writes to take place on the "primary" server, but any of the "secondary" servers can support read requests.

When considering a large-scale distributed web application, there are now a large number of individual systems that are executing in parallel or in sequence that may influence the fingertip-to-eyeball performance experienced by the user. Compared to the traditional web application, there are significantly more potential causes of a performance problem in a distributed web application.

2.4 Events in Distributed Systems

There has been a significant amount of research and discussion about the challenges of capturing events in a useful way in a distributed system. Lamport discusses some of these problems [31]. With respect to web applications, the two challenging problems are time synchronization and establishing Lamport's "happened before" relationship. Since half of the application (client-side) is running on systems that the software developers have no control over, an event capture system cannot make any assumptions about the system clock being even approximately correct (i.e. there is no guarantee that they are running any kind of time synchronization daemon). As for the "happened before" relationship, the HTTP protocol is stateless and asynchronous, often with many requests happening in parallel.

Past the theoretical constraints, there are also additional practical constraints. For example, the developers do not have any remote control over the environment that the JavaScript code runs in. While many of the tracing techniques explained in Section 2.7 would serve well for JavaScript, these techniques require the ability to install invasive tracing into the JavaScript interpreter on the client-side, which is typically not possible.

In Lamport's framework, there are three conditions on the "happened before" relationship (denoted by "$\rightarrow$"). For events a and b:

1. If a and b are events in the same process, and a comes before b, then $a \rightarrow b$
2. If a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \rightarrow b$
3. If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$

To establish this relationship between processes, there are different techniques with varying trade-offs. Lamport Timestamps establish a "logical clock" (denoted C), such that if $a \rightarrow b$, then $C(a) < C(B)$ [31]. Fidge [17] and

[10] http://www.postgresql.org/docs/9.1/static/high-availability.html

Mattern [37] describe Vector Time, an improvement on Lamport Timestamps that is specifically designed to address the fact that Lamport Timestamps can result in many different partial orderings.

Instead of having a single logical clock, Vector Time establishes a logical clock for each process. This model is an improvement over Lamport Timestamps because it establishes a per-process view of global state that can be used to determine a more accurate model of potential causality. Downsides of Vector Time are that the size of the vector grows as the number of participating processes grows (and the vector is attached to every transmitted inter-process message), and that all processes must be aware of the other processes to establish a consistent vector.

More recently, TrueTime [12] does not seek to establish a consistent logical clock, but rather seeks to solve the problem of establishing a consistent global timestamp for a distribute data store. TrueTime timestamps are based on an error interval on the timestamp of event e denoted $[earliest, latest]$, such that $earliest < t_{abs}(e) < latest$. To establish such an error interval, there are a number of highly precise time sources (GPS, rubidium clocks, etc) installed at each data centre; daemons poll multiple time sources and use Marzullo's Algorithm [36] to reject misbehaving time sources. One downside to this technique is that it requires a significant investment in high-precision timekeeping equipment. It does establish a robust time ordering of events, and can be used for globally consistent distributed transaction synchronization, but does not establish causality relationships between transactions.

Since the majority of the processes involved in web applications use a request/response model (possibly asynchronous) with few inter-request interactions, instead of a more general set of computations, it is more important to determine strict causality instead of a global partial ordering of events in the entire system. MT-WAVE uses the approach from X-Trace [18], where events are assigned unique identifiers and messages from one process to another are annotated with the identifiers from events generated specifically for the specific messages (breadcrumbs). This does have the downside that no global event ordering is established, but provides precise causal traces of every event generated by a specific request.

2.5 Analysis Strategies

Gregg [20] highlights the fundamental problem facing those wishing to improve application performance: "performance issues are often analyzed randomly: guessing where the problem may be and then changing things until it goes away." Gregg's USE Method formalizes the high-level approach to diagnosing performance problems: Utilization, Saturation, and Errors. This approach is a highly black box approach, where you look at device statistics (e.g. CPU, Network, Disks, etc) to determine which subsystem is likely to be causing performance problems.

Millsap *et al.* [40] have taken a different approach, focused primarily on the response times of individual user actions. They summarize their method as "Work first to reduce the biggest response time *component* of a business' most important user action." (emphasis added). This technique relies on a version of Amdahl's Law to find the best place to focus on performance improvement: any improvement can only improve response time by the fraction of the original response time it originally took. This requires focus to be placed on the component that currently makes up the largest fraction of the response time.

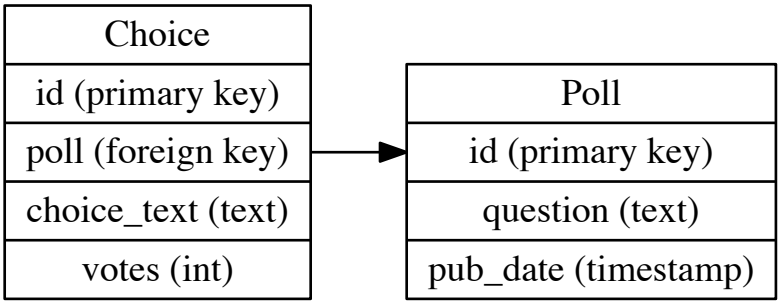


Figure 2.3: Data Model for the Django Polling Application.

Purdy [44] reinforces Millsap's response time argument and expands it to "track down any application's transaction latency from the user's request event to the ultimate screen paint." He argues that a set of tools should be used to track the complete path of the transaction, with latency breakdowns for each component of the transaction. Purdy's argument is essentially that we need to combine the approaches of Gregg and Millsap to effectively solve system performance problems.

2.6 Web Application Implementation

Consider the Django Tutorial[11] as an example of the specific internals of a web application. While this is a simple example, it demonstrates many of the concepts and structures used in web applications. In the tutorial, the authors are creating a Polling application: users vote for specific choices in each poll and an administrator can create new polls.

Figure 2.3 outlines the data model of this application. Each Poll object can have many Choice objects. This is implemented using the Django Object-Relational Mapper,[12] which takes Python-based model definitions and transforms them into SQL tables and queries.

Next, the tutorial outlines the creation of a Django View. This is the piece of code responsible for handling an incoming HTTP request and returning a response. First, an entry must be made in the URL mapper, which parses the incoming HTTP request and determines which view should execute to handle the specified URL. In Figure 2.4, the request for "/" (the root URL) is mapped to the function "index" in the "polls" module.

The basic view described early in the tutorial simply returns static content. A more useful view accesses the models, and uses the template system to generate HTML. The models, in turn, execute an SQL query to populate all of

[11] https://docs.djangoproject.com/en/1.6/intro/tutorial01/
[12] https://docs.djangoproject.com/en/dev/topics/db/

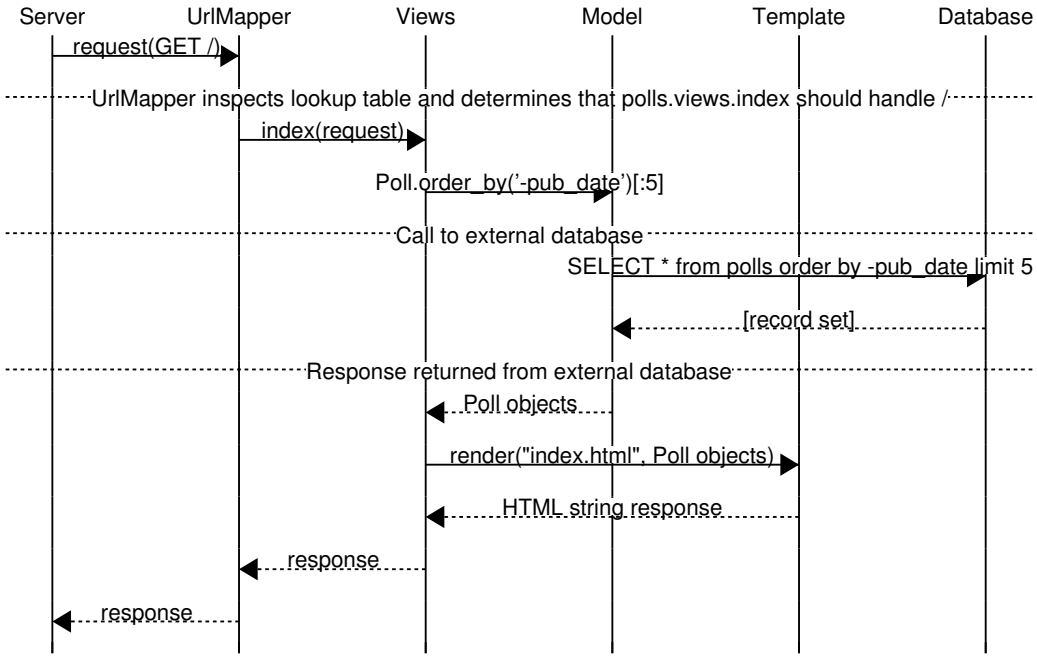


Figure 2.4: Incoming request for the Django Polling Application.

their fields. Figure 2.4 shows the interactions between these different modules, including the call made to the external database.

2.7 Tracing Techniques

2.7.1 Blackbox Tracing

Blackbox tracing techniques treat the application as an opaque system and perform tracing solely based on externally-visible markers (e.g. network traffic or performance counters). These techniques do not typically provide any insight into the internals of the systems being monitored.

Magpie [3] constructs a trace of system activity by recording per-component low-level events and using a per-application schema to associate these events with component activities. Unlike many of the other systems below, Magpie does not propagate request-specific identifiers through each application; instead, Magpie performs a temporal join across the events to reconstruct an approximation of the event causality in the system.

Aguilera *et al.* [1] record message-level events and reconstruct full-system traces using signal processing techniques. Their system is entirely black-box: it does not require any code modification, nor does it require any application-specific information. By design, this system is only suitable for debugging latency between the system components, without providing insight as to the possible cause of the latency inside the slow components. For systems that require inferred causality, the algorithms presented could be very useful.

Chen *et al.* have created Pinpoint [10], a system designed for large-scale fault detection in dynamic systems. Large-scale, in this context, means that Pinpoint is looking to identify failing components across multiple client requests. To accomplish this, they assign unique IDs to each incoming request and propogate these IDs by modifying the middleware on each component. This means that request dependency is directly captured, but the individual components are still treated as black boxes. Their data analysis techniques are incredibly interesting–they use data mining techniques to identify different clusters of failing requests. In large systems, it is not reasonable to assume that all failing requests have the same root cause; these data mining techniques cluster failed requests based on the components involved and provide insight into different unrelated failures that may be occurring simultaneously.

Stardust [56] is a monitoring system for a large-scale distributed storage system. While this is not directly related to web applications, they do encounter similar concerns and difficulties. Stardust, like Pinpoint, traces system-to-system causality using unique IDs (referred to as "breadcrumbs"). To add this system modification without changing the client APIs, they have modified the RPC layer to automatically propogate these breadcrumbs. Stardust, like several other systems, is primarily focused on attributing request latency to specific machines or services, without actually identifying which part of the code is causing this latency.

For monitoring black-box components, without requiring any component modification, Koskinen and Jannotti developed BorderPatrol [29]. This system monitors the inputs and outputs of black-box components to infer request flow through the system. While this monitoring is done without requiring any source-code modification, it uses the somewhat invasive technique of Library Interposition (that is, replacing the dynamically-linked libraries used by the black-box components). Additionally, BorderPatrol works by understanding the input and output protocols at each black-box, which must be user-specified. Like other black-box monitoring systems, BorderPatrol can attribute portions of request latency to each component, but provides no hints about which code inside the black-boxes may be causing the undesirable latency.

Instead of integrating with the web application, mBrace [58] is a backend profiling tool that integrates with the layers surrounding the application. Specifically, mBrace has an Apache plug-in (which sits in front of the web application) and a MySQL plug-in (which sits behind the web application). By using the Performance Monitoring Unit (PMU) available in modern processors, mBrace gathers low-overhead performance statistics about the running application. Like Stardust, Pinpoint, and X-Trace, unique "action identifiers" are assigned to each incoming request. To account for different user actions, mBrace requires the performance analyst to provide a list of URL-to-action mappings.

Whodunit [8] takes a significantly different approach to causal tracing than the other systems. Whodunit is sufficiently general that it can capture request flow (called "transactions" in this work) through a shared-memory system with no modification at all. To accomplish this, they identify critical sections based on the locking primitives and run the critical sections within an emulated CPU. In this emulator, they can watch every byte of memory written and read. This technique is likely not particularly relevant for web applications, where message flow is usually achieved with RPC instead of shared memory, but it is a fascinating and novel technique for instrumenting otherwise black-box systems.

Krushevskaja and Sandler [30] start out by outlining their two key requirements/assumptions: First, call trees are hard to construct; treat services a black boxes and achieve generality by ignoring internal semantics. Second, the reasons for latency variations must be understandable by humans. Their approach is based on the analyst coming up with a set of "potentially related attributes" (some examples are request size, response size, service name, request origin, CPU load, IO load, etc). Based on these attributes, requests get classified into vectors $r_i = (f_1^i, f_2^i, ...f_m^i, \lambda^i)$, where f_j^i is feature j observed for request i, and λ^i is 1 if the observed latency is within the region of interest (e.g. if we're trying to determine the cause of increased latency, λ^i would be 1 for the high latency requests). Using these vectors, they apply the class of machine learning algorithms called "Multi-dimensional F-Measures." In practice, this ends up being a complex and computationally challenging technique that may require more investigation.

2.7.2 Breadcrumb/Metadata-based Tracing

Fonseca *et al.* have developed X-Trace [18], a system designed for doing event capture in distributed systems. X-Trace consists of an event-capture server, an over-the-wire event format, local event forwarding proxies, and application-specific event generation modules. Each traced application is slightly modified to produce informative events; the events each have a reference to the previous event in the causality chain. These events are sent to the local proxy, which will forward them to the centralized X-Trace server. At the central server, the events from all of the different application components are grouped by a Task Identifier (all events produced for a specific application task will have the same identifier). When all of the events have arrived from the different components, the event causality chain can be reconstructed to obtain a trace of all the systems that were involved in processing a request, including the message flows between them.

Dapper [54] is the distributed systems tracing infrastructure in use at Google. This work highlights the main research problem that I am addressing: "Understanding system behaviour in [complex distributed systems] requires observing related activities across many different programs and machines." While Dapper is similar to X-Trace in many ways, the authors have made a number of different design choices than those from X-Trace. A significant difference is that X-Trace uses an arbitrarily-structured Directed Acyclic Graph (DAG) for its event structure, while Dapper uses a tree-and-span structure. The tree-and-span structure is based on nested Remote Procedure Calls (RPCs); the spans represent the activity in each RPC, and the tree represents the nesting structure of the RPCs involved in the overall activity. Another significant difference is that Dapper strives for application transparency–although an application can be "Dapper-aware," all Google applications have access to the Dapper framework because it is embedded transparently in the RPC library. A final significant difference is that Dapper is designed fundamentally for sampling; instead of tracing every request, Dapper is designed to only trace a small subset of requests.

2.7.3 Back-end Profiling

Back-end profilers typically instrument a single component and determine which lines of code are contributing the most latency. Generally these tools are specific to each run-time environment, but can be used for any application in that environment. These tools are not particularly novel and have been in use for many years.

Graham *et al.* introduced gprof [19], the canonical example of a profiler. Developed originally in 1982, gprof is still in use today as a staple profiling tool for compiled executables. Gprof has two light-weight measurements: function invocation counts and statistical sampling of function run time. Invocation counts are simple: each function records the number of times it has executed. Statistical sampling is slightly more elaborate: periodically, the program is temporarily interrupted and the value of the program counter is recorded. By tracking the program counter (currently executing instruction) periodically, gprof produces an estimate for how much time is spent in each function.

cProfile[13] is the standard Python profiler. Python provides instrumentation hooks through the sys.settrace[14] facility. cProfile can be used to get function call statistics (total time, cumulative time, number of calls) for Python code. This is a relatively heavy-weight approach for analyzing Python code and carries a significant overhead. Unlike gprof, which only does statistical sampling, cProfile measures function time exactly, by recording the entry and exit times of each function call.

For PHP, XDebug[15] is the standard profiler for gathering trace data and function call statistics. XDebug is installed as a site-wide PHP plug-in, and is activated by sending specially-formatted HTTP requests to the URL to be profiled. When profiling, the output is saved onto the serving machine, to be collected offline for post-processing and evaluation. XDebug also has a built-in remote debugger; a developer can debug PHP scripts running on a server from their own workstation. This a versatile tool for understanding the execution of single PHP requests, but does not provide any aggregate information about the series of HTTP requests involved in a single page load.

Erlingsson *et al.* [16] take a different approach to injecting instrumentation into runtime environments. Instead of having the runtimes explicitly load the instrumentation, they take advantage of the Windows Hotpatching Infrastructure to dynamically inject the Fay Dispatcher into the executable at load time. This is a facility built into Windows that provides an interface to intercept function invocations and returns. This allows them to directly measure the execution times of the instrumented functions with minimal overhead.

2.7.4 Front-end Profiling

Traditional front-end tracing tools integrate with a web browser and provide detail about the HTTP requests involved in a page request. These tools do not provide any information about back-end performance other than the total time required to service the request.

Firebug[16] is a general-purpose Firefox plugin for doing web site and application debugging. It has a number of non-performance-related features that assist in viewing and debugging document structure and visual appearance. Additionally, it has a few built-in performance-related features. The Network Monitor instruments the HTTP requests involved in loading a page. For each request, you get a measurement of how long DNS resolution took, how long it took to send the request, and how long the server took to send a response. The Network Monitor is implemented using the same Firefox instrumentation hooks that the MT-WAVE Firefox plugin uses (see section 3.3.4). Firebug also includes a

[13] http://docs.python.org/library/profile.html
[14] http://docs.python.org/library/sys.html#sys.settrace
[15] http://www.xdebug.org/
[16] http://getfirebug.com/

JavaScript debugger and profiler that can be used to determine some aspects of front-end web page performance. This profiler can only be used to profile the code that executes in the browser; it has no facilities to monitor any back-end behaviour.

Web Inspector[17] is the built-in front-end debugging tool built into WebKit, the underlying technology in Safari and Chrome. Web Inspector provides essentially the same functionality as Firebug, but does not require a plugin to be installed. You can monitor incoming network requests, debug JavaScript (front-end) code, and inspect document structure and visual appearance. Like Firebug, it has no back-end integration.

For Google Chrome, Speed Tracer[18] is a plug-in that instruments the performance of the browser; this is primarily focused on the activities performed by the browser. Like Firebug and Web Inspector, it can be used to monitor network activity; the additional functionality involves instrumenting the browser itself. Speed Tracer monitors the behaviour "behind the abstraction" of the browser and reports how much time is spent in various HTML rendering activities. These results can inform decisions about how to modify the HTML structure to allow for faster rendering.

Y-Slow[19] is an interesting tool; it is an extension to Firebug that automatically makes performance recommendations based on a set of best practices (decided by Yahoo!). To do this, it looks at the results of the Firebug Network Monitor and applies heuristics to the sequence of HTTP requests. Based on these heuristics, Y-Slow will determine HTTP requests that may be slowing down the page load. For example, most browsers place a 6-connections-per-host limit on HTTP traffic. If Y-Slow detects that you are loading 30 image files from the same host, it will suggest that you spread those images onto separate hosts, so that the 6-connection limit will no longer prevent the images from being downloaded in parallel.

Wang *et al.* [60] take the in-browser tracing concept further. Instead of just using a plug-in that takes advantage of the built-in performance profiling functionality of a browser, they directly instrument the core of WebKit to add additional tracing. Using this enhanced tracing data, they build a model of real-world page load performance based on resource dependency and validate this model using Critical Path Analysis to predict page load performance and to determine the potentially highest value optimizations (i.e. bottleneck analysis).

WebProphet [33] is the original work that inspired Wang *et al.*'s solution. Unlike in Wang's work, WebProphet does not do in-browser instrumentation but rather instruments and perturbs packets at the network level to do resource dependency extraction. By delaying certain resources, they build a graph of which resources delay the loading of other resources. By operating at the network level, they have the ability to perturb more than just HTTP requests; they can affect the DNS name resolution process, TCP connection setup times, etc.

AjaxTracker [32] is a tool designed for modelling web application traffic. Unlike most of the other Front-end tools, AjaxTracker does not hook into the web browser at all. This tool, instead, captures web application traffic on the local machine using packet sniffing; all client-server interactions are captured by capturing the TCP streams.

AjaxScope [28] is considerably different than the other front-end tracing tools. Instead of integrating with the browser or performing HTTP traffic sniffing, AjaxScope adds dynamic instrumentation to JavaScript code before send-

[17] http://trac.webkit.org/wiki/WebInspector
[18] http://code.google.com/webtoolkit/speedtracer/
[19] http://developer.yahoo.com/yslow/

ing it to the client; instead of integrating with the client, it modifies the code that the client will be executing. Once this code is executed in the client, the collected instrumentation data is buffered and then logged back to the server using an HTTP POST request.

Similar to AjaxScope, AppInsight [45] does dynamic instrumentation on MSIL (the intermediate language that is part of the Microsoft .NET runtime) for performance evaluation of mobile applications. Like the concept of fingertip-to-eyeball instrumentation, AppInsight is focused on "user transactions," which begin with manipulation of the user interface and end with the completion of all synchronous and asynchronous tasks that resulted from the UI interaction.

2.7.5 Client-side Real User Monitoring

The goal of Real User Monitoring (RUM) is to monitor application performance from the perspective of real users, without necessarily identifying the underlying causes. While this captures significantly less data than other techniques, the primary benefit is that it captures real-world performance data without requiring any client-side modification.

Modern web browsers have the NavigationTiming API[20] for accessing coarse page-load timing data. Although this data provides very little information about which specific parts of the page are contributing latency, it can be augmented with other application-specific instrumentation to provide actionable statistics. The `window.performance.timing` API is used in the evaluation of MT-WAVE overhead, since it incurs no additional overhead; the browser is going to maintain these measurements whether we ask it to or not, so user-perceived performance is not affected if we retrieve this data after the page has loaded. This API provides a series of timestamps corresponding to the start of the page load, the moment when the page becomes interactive, the moment when the page load is complete, and several others.

Steve Souders' Episodes[21] framework is an open source implementation of RUM that is very simple to drop into an existing application. This was used in several experiments to verify that MT-WAVE's performance measurements correspond with those measured by existing techniques. With Episodes, the developer manually adds small pieces of instrumentation code to capture the timing of specific events that happen in the browser. After the page has loaded, this event timing is sent to the web server by using a "beacon": a URL is crafted that contains all of the timing information that ends up in the web server's log files. This data can be extracted off-line to build a performance profile.

aja: The page with these comments was folded over in the scan and I can't read the comments

Meenan [38] is a more recent summary of the available techniques developed as part of the W3C's Web Performance Working Group. This work highlights the need for per-resource timing information and provides code examples demonstrating how to retrieve this data from modern browsers.

2.7.6 Analysis and Comparison Techniques

Sambasivan [49] looks at algorithms for comparing before and after traces. This work discusses two key techniques for comparing request flows. First, they produce a canonical identifier for each unique trace graph (i.e. traces with all of the same method calls end up with the same identifiers) by establishing a total ordering based on a depth-first tree

[20] `https://dvcs.w3.org/hg/webperf/raw-file/tip/specs/NavigationTiming/Overview.html`
[21] `http://stevesouders.com/episodes/paper.php`

traversal and breaking ties alphabetically. This allows them to easily identify traces with identical callgraphs. Then, after ordering the traces, they use the Kolmogorov-Smirnov Test to compare the empirical cumulative density functions (ECDFs) of the traces to identify abnormal traces without needing to make assumptions about the distributions of function call timings. Like Sambasivan, Wang [60] also uses Cumulative Density Function Plots to directly compare before and after performance; their work contrasts CDFs against P-values.

Mann *et al.* [35] outline several analysis techniques with varying degrees of effectiveness. To start, they outline simple bounds for the model: purely sequential children and purely parallel children. For modelling techniques, the first and least effective is linear regression; this builds a model that does not explicitly encode any information about the child-parent relationships between function calls. LR is a useful baseline but fails to generalize into a useful predictive model, especially when parallelism is introduced. Next they look at Critical Path Analysis: an algorithm that attempts to identify the subset of children such that reducing the child's latency reduces the parent's latency. They determine that CPA also gets brittle as parallelism in the call graph increases.

There are several novel analysis contributions in this work. First, there is the Invocation Graph; this does not explicitly encode causality but builds a graph $G_I = (M, E_I)$ such that $(m_i, m_j) \in E_I$ iff m_i finishes before m_j. This results in a graph that approximately encodes a sequential execution flow from potentially parallel method calls. This graph may have spurious constraints due to scheduling of parallel calls (they may have ended up executing such that one finished before the other started, even though they were invoked as parallel children of the parent). To prune these spurious constraints and build a more accurate model, several traces are gathered and combined. An elaboration of this is the Execution Flow Graph, which is similar but $(m_i, m_j) \in E_I$ iff m_i finishes before m_j *starts*. Using the Execution Flow and latencies, they compute an "Effective Critical Path" that finds the longest blocking path through the graph.

Finally, they consider Nearest Neighbour Flow, which clusters traces that have identical Invocation Graphs. Unlike in the work of Sambasivan and Wang, NNF models latencies as Normal and calculates a mean and standard deviation. When abnormal latencies are detected, NNF can be used to estimate which flow has the highest probability of generating the observed latency; analysis of this flow can provide a good starting point for determining the root cause of latency problems.

2.8 Summary

The web started out as a primarily content-based medium; users would go to a website to view mostly static content. Clicking a link on a web page loaded a new web page. As the web evolved, websites became interactive and not every click resulted in a new page loading; the content would change in-place in response to user actions. As this interactivity continued to evolve, web applications began to appear: fully interactive pages with significant amounts of code that executes on the client, with the server being used primarily for data storage and processing. One of the key features of web applications is user-generated content: instead of simply presenting information, web applications allow users to create and upload their own data.

There are a number of parts to a web application. HTTP is the underlying protocol for communication between

the web browser and the server, HTML and JSON are the main data transfer formats between client and server, and JavaScript is the main programming language used in the web browser. Server-side there are many different programming languages that can be used; the only requirement is that there is a library available so that the language can communicate over HTTP. As applications grew in popularity, they could no longer be served using a simple single-server architecture. Modern large-scale applications often consist of large clusters of servers, including integrated third-party services (for example, social network authentication).

Performance tools for distributed computing and applications have been studied for a long time, but this research has not made its way into tools targeted at web applications. There are many existing tools for analyzing server and application performance, and some tools for analyzing web browser performance, but there are very few tools that provide an integrated view of client-side and server-side web application performance. MT-WAVE is a combination of existing tools and techniques chosen for effective multi-tier tracing for web applications.

CHAPTER 3

MT-WAVE & DISTRIBUTED TRACING

3.1 Chosen Tracing Techniques

MT-WAVE combines a set of existing tracing techniques to effectively perform multi-tier tracing for web applications. Breadcrumb tracing is used to make the connection between each tier. Each tier generates tracing events and forwards this metadata to X-Trace for further processing and aggregation. In the browser, a Firefox plug-in is used to fully capture all of the HTTP requests and to inject the breadcrumbs. For instrumenting each tier, several different techniques are used: application framework modification, script interpreter instrumentation, and source-code rewriting. The following subsections provide more detail about the implementation of each piece.

3.1.1 Breadcrumb/Metadata-based Tracing for Event Aggregation

In Chapter 2, two general techniques for event tracing and aggregation are discussed: Blackbox Tracing and Breadcrumb Tracing. After considering the trade-offs and applicability of the two techniques, Breadcrumb Tracing was chosen for MT-WAVE.

Blackbox Tracing's most significant advantage is that it is application-agnostic and can run without access to the application's source code. This is exceptionally useful for many environments (e.g. in a data centre where you are running code that you haven't written). There are, however, three main disadvantages: most blackbox techniques perform some form of heuristic or probabilistic tracing, there is often a computationally-significant post-processing step, and in the end, the results can only attribute performance problems with component-level granularity.

Breadcrumb Tracing has roughly an inverse set of advantages and disadvantages. It is not application-agnostic and usually requires access to the application's source code. Breadcrumb Tracing results in exact execution traces, instead of heuristic- or probability-based traces. It requires minimal post-processing, and it can attribute performance problems to specific lines of code.

For MT-WAVE, the target audience is web developers who are investigating the source of performance problems in their own applications, thus there is availability of the application source code. Additionally, the platforms and frameworks investigated run on interpreted programming languages, where the application code executes inside an instrumentable virtual machine. This means that Blackbox Tracing's key advantage is mostly unnecessary to the goals of MT-WAVE. The additional advantages of Breadcrumb Tracing (exact traces, minimal post-processing, specific source attribution of performance measurements) further reinforce this choice.

X-Trace was chosen as the Breadcrumb Tracing platform to sit underneath MT-WAVE for several reasons. Importantly, X-Trace is available as an open-source project under the permissive BSD license. This ensured that any modifications required for the research was possible. The simple wire format for events ensured that integration with different application frameworks would be straightforward. Finally, this package has sufficient throughput to support the experiments in this work.

3.1.2 In-Browser Capture

For detailed in-browser performance measurements, there are a number of instrumentation hooks that can be used. While this is not a technique that can be used all the time (it requires additional software to be installed on the client side), it can be used by developers who are trying to diagnose performance problems in their own web applications.

One of the most useful in-browser hooks gives access to all of the network events: requests that get queued, connection times, data transfer rates, etc. While these are not the direct measurements required for measuring fingertip-to-eyeball performance, these can provide significant insight into the browser-perceived performance of the server-side of the application. Additionally, these hooks can be used to inject breadcrumbs into the HTTP requests to enable correlation of client-side events and server-side events.

Firefox, in particular, has a very rich set of instrumentation and observation points[1] that are used for MT-WAVE. The "http-on-modify-request" hook is used to determine when a request is started and to modify the outgoing request headers with the appropriate X-Trace Metadata, and the "http-on-examine-response" hook is used to determine when a response has been returned. Chrome, too, has a thorough set of observation and modification points,[2] which were not available when the in-browser capture plugin was written. As such, Firefox is the only browser that MT-WAVE currently targets.

3.1.3 Application Framework Modification

An invasive, but effective, technique for capturing events in the application is to directly modify the application source code to generate events at appropriate times. This requires quite a bit of developer thought and intuition to determine the appropriate places to add instrumentation. In the event that instrumentation is not added in the right places, the resulting trace data may be useless. Done correctly, though, this technique can provide precise information about which application events are occurring.

A less invasive and less thought-heavy technique is to modify the application framework. Instead of adding instrumentation into the application itself (and requiring application-specific changes each time instrumentation is required), instrumentation gets added at key points in the framework code that is shared between many applications (e.g. Django). While these will be less precise than application-specific modifications, there are many common tasks that can be instrumented easily (e.g. start of request handling, database queries, HTML rendering, etc). One significant benefit of this technique is that the instrumentation can be reused for many different applications.

[1] https://developer.mozilla.org/en/docs/Observer_Notifications
[2] https://developer.chrome.com/extensions/webRequest

A combination of these two techniques is used in the Python-based case studies in Section 4.2.5. A small library for parsing and generating X-Trace metadata in Python was created and the Django framework was then modified in a few key places to capture critical events (e.g. in the SQL library, to emit events for every SQL request made from the framework to the database).

3.1.4 Script Interpreter

Even more general than application or framework modifications is Interpreter instrumentation. Many interpreters provide hooks for tracing an entire application; for example, Python and PHP have hooks that can get called at the beginning and end of every function invocation. This provides much more detailed insight into the execution behaviour of the application, but often carries a more significant performance cost.

This technique is used in the Joomla (Section 4.2.2), Magento (Section 4.3.1), and Wordpress (Section 4.3.2) experiments. To minimize the in-process overhead, a very simple PHP plug-in was used to capture the beginning and end of function invocations. The raw function invocation data is written to a Unix domain socket, which is read asynchronously by a process that converts the raw data into X-Trace metadata and forwards the metadata to the X-Trace server running elsewhere.

3.1.5 Javascript Instrumentation

JavaScript is a very challenging language to instrument, because it usually executes outside of the administrative control of the application owner. For an organization that wishes to improve the performance of their web application, understanding the general JavaScript performance (as perceived by their users) is a critical issue. To solve this problem, one could use the application-specific instrumentation technique outlined above, but it would be ideal if a more general technique (like script interpreter instrumentation) could be used.

Schrock [50] outlines a manual technique for adding in JavaScript instrumentation. This work is focused on applying wrapper functions to methods within the existing JavaScript codebase for catching errors and reporting them back to the application's developers. This work describes a mechanism provided by the JavaScript interpreter embedded in many browsers: the `window.onerror` callback. While this work describes a good practice, it is focused on error reporting and not on performance evaluation.

One approach to solving this problem is to do dynamic JavaScript rewriting. This is approximately the technique is described by Kiciman and Livshits [28]. Before the JavaScript code is sent to the browser, it is passed through a filter that automatically adds instrumentation to it. This is essentially an automatic application of Schrock's technique; instead of manually applying wrapper functions, all functions are modified to add performance and error trapping code. MT-WAVE uses a small HTTP proxy that identifies requests for JavaScript files and rewrites them to include function invocation logging. When a browser is configured to send requests through the proxy, most content flows through unmodified, but JavaScript code is parsed and rewritten before being returned to the browser.

3.2 MT-WAVE

3.2.1 Introduction

MT-WAVE is the combination of the technologies and techniques described above. It is a collection of event tracing tools (Python, PHP, Ruby, and JavaScript), an event model for capturing and processing the code execution in web applications, an event logging tool (X-Trace), and an event visualization system. These tools are combined together into a comprehensive web application performance monitoring system that can be used to effectively determine the root cause of performance problems in web applications.

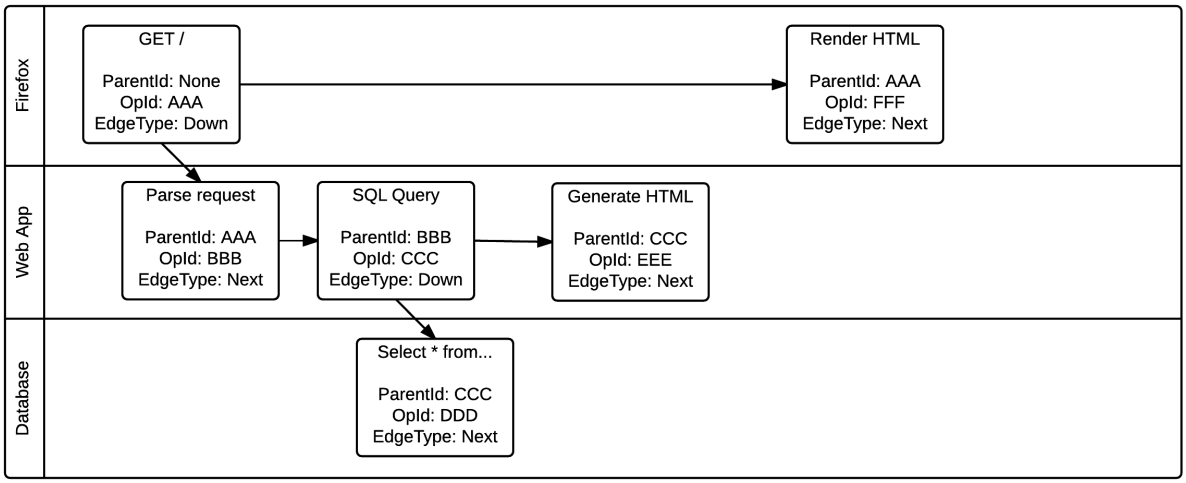
This system is designed to help performance analysts answer a specific question: which code in a system is causing the largest user-perceived latency? While there are many possible performance metrics available (service latency, throughput, etc.), this so-called "fingertip-to-eyeball performance" is the only metric that end-users can directly perceive. By addressing this metric, MT-WAVE gives performance analysts the tools needed to directly improve the end-user experience.

3.2.2 Event Model

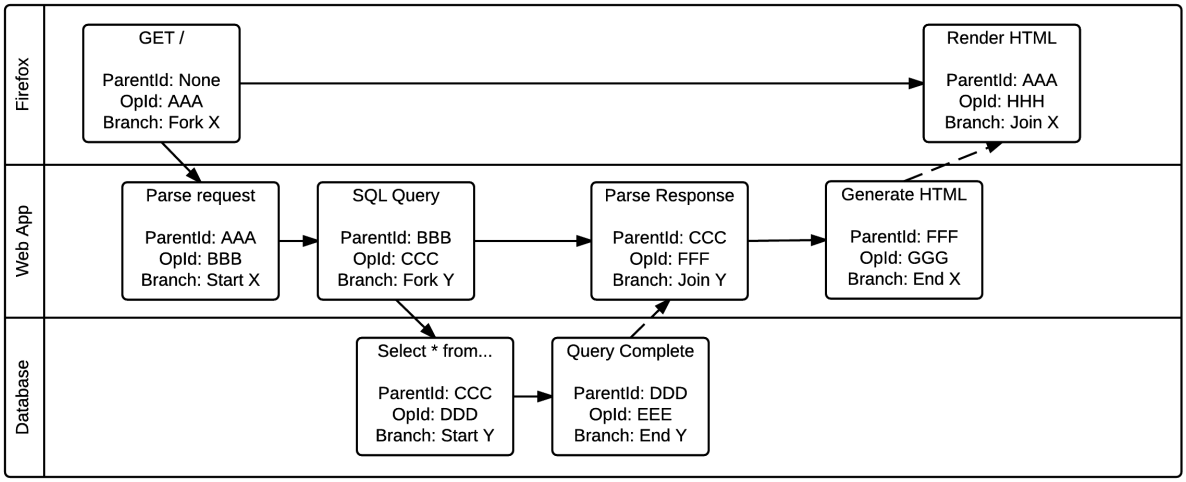
The baseline X-Trace event model is described by Fonseca *et al.* [18]. In this model, each event is identified by three subfields (*ParentID*, *OpID*, and *EdgeType*), and two operations (*pushDown()* and *pushNext()*). The ID fields are both 32-bit values; the *OpID* is randomly generated for each event, and the *ParentID* is set to the value of the *OpId* of the causal parent event. The *pushDown()* and *pushNext()* operations behave identically with regards to the ID fields, but set a different *EdgeType* value ("down" and "next", respectively). Along with other metadata fields, these fields are transmitted in each request made to different tasks.

These fields and operations result in a "task tree" that exactly captures the causality relationships between the different events. Within a given task, the *pushNext()* operation is used for each event, and when sending a message to another task, the *pushDown()* operation is used. When reassembling the task tree, the resulting *EdgeType* fields result in a layered task tree that shows event causality both internally to each task and between the tasks. Since this model is designed with metadata transmission only occurring at request time, there is no backward event synchronization inherent in this model; event causality can be determined precisely, but information about request completion must be determined implicitly.

To precisely determine both the beginning of a request and its completion, the MT-WAVE event model augments the X-Trace event model by adding a *Branch* field. This field consists of a "type" subfield ("fork", "join", "start", and "end"), and an "ID" subfield, which is a unique 32-bit identifier. When a client makes a request, it emits an event with a "BranchFork" event, and when it receives the response to that request, it emits a "BranchJoin" event; likewise, the server emits a "BranchStart" event at the start of request processing and a "BranchEnd" event at the end of the request. See Figure 3.1 for a comparison of the event graphs.



(a) X-Trace event tree without the *Branch* field.



(b) MT-WAVE event trace with the *Branch* field.

Figure 3.1: Comparison of event traces with and without the *Branch* field.

3.2.3 X-Trace Extensions

X-Trace is a generic event capture framework that was not designed specifically for web applications. As MT-WAVE development progressed, it became clear that there was additional functionality that could be added to X-Trace to provide an easier experience for this specific problem domain. Additionally, there were performance problems when writing large numbers of events; the sustained throughput was acceptable, but burst performance was found to be lacking.

3.2.3.1 JSON Events

The basic X-Trace event format is very simple: text-based "Key: Value" pairs, one per line, with a blank line separating events. In many environments, this is an ideal format: it is simple to generate on the producer side, and simple to parse on the consumer side. In JavaScript, though, JSON is the easiest format to generate and consume. A call to `JSON.stringify` will convert a JavaScript object to JSON, and a call to `JSON.parse` will parse a JSON-formatted string back into a JavaScript object. All common backend web frameworks have a similar method for parsing and generating JSON. Figure 3.2 shows a comparison between plain-text and JSON-formatted X-Trace events.

```
Agent: HttpRequestObserver
Edge: 0000000000000000
Epoch: 1.409516816029E9
Label: Load Document request for: http://172.17.0.3/
Tag: 20140831-142115
Timestamp: 1.409516816029E9
Title: /
X-Trace: 1980f9ce54fda4c5030d5497a3bcc35163
```

(a) Plain-text X-Trace event.

```
{
  "Agent": "HttpRequestObserver",
  "Edge": "0000000000000000",
  "Epoch": "1.409516816029E9",
  "Label": "Load Document request for: http://172.17.0.3/",
  "Tag": "20140831-142115",
  "Timestamp": "1.409516816029E9",
  "Title": "/",
  "X-Trace": "1980f9ce54fda4c5030d5497a3bcc35163"
}
```

(b) JSON-formatted X-Trace event.

Figure 3.2: Comparison of plain text and JSON-formatted X-Trace events.

JSON supports arrays for serializing multiple objects, e.g. `[...obj1..., ...obj2...]`. In the case of an X-Trace event stream, parsing JSON arrays is not ideal: the stream may be quite large, and the parser will return an error if the array is incomplete. This means that the entire array must be buffered as text before the conversion to objects can take place; not only does this introduce delays, but it also requires a significant amount of memory to store the text.

The X-Trace JSON event streaming extension works around this by using a custom streaming JSON. This parser does not require a complete array of events; instead, it invokes a callback function every time a complete JSON object has been parsed. This means that the parsed objects are passed on to the event persistent module as soon as they have been parsed instead of waiting until the end of the event stream.

3.2.3.2 X-Trace Performance - SQL Transaction Batching

Apache Derby[3] is an embedded Java database that is used internally by X-Trace. In the initial MT-WAVE experiments, X-Trace event throughput performance was found to be lacking. This was not an issue during the Python experiments, since manually inserting trace statements does not result in a large event load. During the PHP experiments, where every function call is traced, it became clear that X-Trace could not process the events quickly enough to prevent unbounded growth in the processing queues. This would eventually lead to memory exhaustion. Until memory was exhausted, there was no indication that the system was struggling to process the events; since the running program is only emitting events and not receiving anything back, the queues successfully hid the processing latency.

This performance problem was traced to an interaction between the Event Persistence module and the Derby database; X-Trace was inserting each event into the database one-by-one as they were received, and the "SQL Insert" performance in Derby was slow enough that it was impeding the event stream.

To alleviate this pressure, the X-Trace Database module was modified to insert events in batches instead of one at a time. Events are stored in a temporary data structure, and every 5 seconds they are committed to the database in a single transaction. This delay was chosen as a trade-off between performance and interactivity: if the delay is too short (favouring interactivity), then the performance gains are significantly reduced; if the delay is too long, performance improves but events do not appear in the database until the delay has expired, which is confusing to the user.

3.3 Tracing Modules

3.3.1 Python

The Python tracing module is the simplest of the tracing modules. It consists of two parts: a library for emitting X-Trace events, and a modified version of Django that emits events at key points in the application's request-handling process. Out of the box, this provides generic tracing for a Django app, with a simple API to add application-specific tracing as well.

The library is a standard Python module that can be imported into Django using the Middleware framework. For the most basic tracing, an application developer only needs to add "import mtwave" to their Django configuration file, along with the X-Trace hostname. For application-specific tracing, the developer adds "import mtwave.trace" to the top of the file that is being traced, and uses a decorator function to add instrumentation. Figure 3.3 shows an example of function decoration: *@mtwave_log* is the decorator function, and is wrapping the *r2r* function. The first two parameters

[3] `http://db.apache.org/derby/`

specify the “Agent” and “Label” fields that will end up in the X-Trace logs, and the “capture” parameter specifies that arguments 0 and 1 should be captured and inserted into the X-Trace log as “TemplateName” and “TemplateContext” respectively.

```
@mtwave_log("Django", "Rendering␣Template",
  capture={ "TemplateName" : 0, 'TemplateContext' : 1 })
def r2r(template_name, template_context, request):
  request_context = RequestContext(request,
    processors=[perms_processor])
  return orig_r2r(template_name, template_context, request_context)
```

Figure 3.3: Decorating a Python function with the MT-WAVE decorator.

3.3.2 PHP

Tracing PHP happens through an interpreter extension written in C. In contrast to the Python tracing module, where the Django library was modified, the PHP extension intercepts the `execute` function in the PHP interpreter, which is called at every function invocation; this results in a full trace of the entire application, including the framework, application, and any standard library calls. Correspondingly, this also results in a significant increase in added overhead.

During interpreter startup, two function pointers are manipulated. PHP uses functions called `zend_execute` and `zend_execute_internal` to execute all of the function calls in a PHP script. The plugin captures the existing values of these pointers and replaces them with `mtwave_execute` and `mtwave_execute_internal`; these replacements wrap the function invocation with appropriate logging, and then call the original execution functions.

When servicing a request, the MT-WAVE plugin first checks for the MT-WAVE HTTP header that is sent by the In-Browser Plugin; if it is not found, it simply aborts the instrumentation process, resulting in negligible overhead in the non-tracing case. This allows the developer to deploy the plug-in to a production server without worrying about affecting non-trace performance. If the header is found, the initialization process is as follows:

- Parse the HTTP header. The header contains the X-Trace taskId, opId, and branchId that corresponds to the request made at the browser.
- Open a Unix Domain Socket to the listening proxy, if necessary. This socket is used to forward trace data to a proxy server that parses the output from the tracing library and converts it into X-Trace events.
- Log the start of the request. This initializes the parsing library in the proxy and prepares it to receive the invocation logs.

The `mtwave_execute` functions are very similar and follow the same pattern:

- Check to see if tracing is active for this request; if not, simply call the original `zend_execute` function and return.

- Extract the function name, source file, and line number if possible. Some internal functions are written in C and do not have an accessible file/line number; in that case, the plugin simply records the function name and the fact that it is an internal function.
- Log the beginning of the function invocation, including a timestamp. This sends a simplified event to the proxy listening on the other side of the Unix domain socket, which will translate the simple format into a full-blown X-Trace Event and forward that event to the X-Trace server.
- Call the original `zend_execute` function.
- Log the end of the function invocation.

The proxy that converts the simplified PHP events into full-blown X-Trace events is a relatively standard design for a forking Unix server. The server binds to a Unix Domain socket and then sits and waits for incoming connections. When a new connection arrives, a client-handler is forked to handle the new incoming connection, and the server carries on waiting for incoming connections and reaping completed child processes.

The child process is designed to accept trace data as quickly as possible. To this end, it does no processing on the incoming data at all until the client has disconnected. All of the incoming data is stored in a buffer; this buffer grows multiplicatively when it fills, to reduce the number of reallocations. This type of data structure would theoretically cause a concern due to memory fragmentation; in practice, a given child process only has one of these data structures, and since the client exits after processing, the fragmentation goes away.

After the client has disconnected, the incoming event stream is processed using an embedded Lua-based processor. The client has already disconnected at this point, so we no longer need to handle processing as fast as possible; this resulted in string parsing code that was significantly simpler that the equivalent C code and still performed very well.

3.3.3 Ruby On Rails

Ruby on Rails tracing is similar to PHP tracing: every function call is recorded, and the event stream is passed through a Unix domain socket to the same proxy that PHP uses. This tracing plug-in is written in Ruby and included in the application configuration, in contrast to the PHP plug-in, which had to be written in C and included in the interpreter/web server configuration. The result is that this is significantly simpler for developers to add to their own application.

The tracing plug-in is designed as "Rack middleware." When a Ruby web application receives a request, it is parsed and then passed to each registered middleware handler before arriving at the application. After the application has processed the request, the middleware is traversed in reverse order. MT-WAVE is inserted at the top of the middleware stack, ensuring that it will execute first and will have an opportunity to clean up at the very end of the request.

This plug-in is an excellent demonstration of how simple it is to add MT-WAVE tracing to a new environment; it was written at the last minute when an opportunity arose to trace a large commercial application. It was implemented the same day the opportunity arose, and immediately found two performance issues (see Section 4.3.3 for more details).

To add this MT-WAVE tracing to a Rails application, a copy of `mtwave.rb` is added to the application, and `config.middleware.insert_before "Rack::Sendfile", "Mtwave::Middleware"` is added to `config/application.rb`.

Tracing in the application is disabled by default, and only enabled when a user has the MT-WAVE Firefox plug-in installed and activated.

3.3.4 In-Browser Plugin

The MT-WAVE Firefox instrumentation system is written in JavaScript and relies on hooks exposed by the browser: the "observer-service" and the "http-activity-distributor". These two services provide us with timestamps for the following events:

- Request is queued and its response is received;
- Beginning of DNS resolution;
- TCP socket connection beginning/establishment;
- Each HTTP header is successfully sent; and
- Browser begins to receive an HTTP response.

These services also provide us with the ability to detect when a new page is loaded, associate HTTP connections with page loads, and add custom HTTP headers to the connection. When a new page is loaded, we generate a new X-Trace Task ID and associate that metadata with the new page. Any HTTP requests that happen with that page are annotated with X-Trace headers that indicate the shared Task ID and unique OpIds that are later used to identify the causality chains.

This extension is exposed as an icon next to the address bar in the browser. Once configured, a developer can enable and disable tracing by clicking on the button. Right-clicking it opens the configuration dialog, which is used to set the address and port of the listening X-Trace server. Other than a check-mark appearing on the MT-WAVE button, there is no other indication that tracing is in progress other than a slight slowdown due to tracing overhead.

3.3.5 Javascript Rewriting Proxy

The rewriting process has two steps: include a tiny MT-WAVE library at the top of the source file, and rewrite the function calls in the original source to call the MT-WAVE instrumentation. The library defines an `mtwave` global object with a few methods. The `mtwave.start` method should be called at the start of a function, and the `mtwave.end` method should be called at the end of a function. These two methods are optimized for performance: all they do is write a small event object to a global `mtwave.events` array. There is also an internal `mtwave.event_sender` method which is called repeatedly during idle time; this method opens an asynchronous HTTP request to flush the events array from the browser back into the proxy for further processing.

3.3.5.1 JavaScript Rewriting Rules

Rewriting JavaScript code is done using the UglifyJS[4] library. The source file is parsed into an Abstract Syntax Tree, the AST is manipulated, and a new source file is generated from the AST. This process appears straightforward at first, but there are several nuances in the JavaScript language that must be taken care of to add instrumentation without generating code that appears valid but is syntactically incorrect.

To start, consider the code in Figure 3.4. This code takes a variable `n`, determines whether it is divisible by two, and either returns `n/2` or `3*n+1`. This function demonstrates the first important nuance in the rewriting process:

Functions may have multiple exit points.

While Figure 3.4 demonstrates multiple return points explicitly, the language also supports exceptions. From an instrumentation point of view, exceptions can be considered as implicit return points that can happen during the execution of any statement in a function. Shown in Figure 3.5, the solution to this is to wrap the entire body of the function in a `try...finally` block, with the call to `mtwave.start()` happening before the block and the call to `mtwave.end()` happening inside the `finally` clause. Whether or not an exception is raised, the `finally` clause is guaranteed to execute before control returns to the caller.

```
function f(n) {
  if (n % 2 == 0) {
    return n / 2;
  } else {
    return 3*n + 1;
  }
}
```

Figure 3.4: Example of a simple JavaScript function.

[4] https://github.com/mishoo/UglifyJS

```
function fx(n) {
  mtwave.start("fx");
  try {
    if (n % 2 == 0) {
      return n / 2;
    } else {
      return 3*n + 1;
    }
  } finally {
    mtwave.end("fx");
  }
}
```

Figure 3.5: Example of a simple JavaScript function wrapped in a `try ... finally` block.

Closures, Scopes, and Hoisting

When a JavaScript function is created, it is assigned a Lexical Environment (Scope) that contains all of the variable names declared within the function, and a reference to the parent Scope; in other words, when a name is used inside a JavaScript function, it may refer to a variable declared in the function, or it may refer to a name that was valid at creation time (in the parent's scope, recursively). A very common idiom in JavaScript is for a function (A) to return a new function (B), where B has captured variables from A's scope. This idiom is problematic when wrapping the body of a function in a try/finally block, because that construct is illegal according to the JavaScript specification.

Variables within a JavaScript function are declared using the 'var' keyword, and may exist anywhere within the function body. If a variable is used that has not been declared as a 'var', the parent scopes (defined at function creation time, not at function execution time) are searched recursively; if the name is not found, it is assumed to be global. Despite the fact that variable declaration may happen anywhere inside a function, the function's Scope (and list of variable names) is determined at creation time, without any dependency on the order or location of the name declarations within the function body.

We take advantage of this to rewrite the body of the function to work around the fact that it is illegal to declare a function inside a try/finally block. Since the scope is determined at function creation time, we can simply extract any sub-function declarations within the body of a function and move them to the top of the function body, before our inserted try/finally block. Figure 3.6 shows the input and output to the function hoisting procedure. We generate a unique name for the function to ensure that it can be referenced later, within the try/finally block.

```
function f(i) {
  var g = 3;
  var h = function() {
    // g refers to g in the
    // parent scope
    return g;
  }
  return h;
}
```

```
function f(n) {
  var mtwave1234 = function() {
    return g;
  };
  mtwave.start("f");
  try {
    var g = 3;
    var h = mtwave1234;
    return h;
  } finally {
    mtwave.end("f");
  }
}
```

Figure 3.6: Hoisting a closure to the beginning of a JavaScript function does not change the semantics of the function.

3.3.6 Proxy Implementation

The rewriting proxy acts like a standard HTTP proxy. To use it for debugging JavaScript performance, the browser must simply be configured to route all HTTP traffic through the proxy. Any requests for content that is not JavaScript will be passed through unchanged; only JavaScript code is inspected and rewritten.

The pseudocode for the proxy is detailed in Figure 3.7. Requests are forwarded to the destination server and the responses are rewritten if they contain JavaScript code.

```
while(true) {
  wait for request
  parse request URL
  if (url == '/mtwave-event-capture') {
    save event stream from browser
    return empty response
  }
  retrieve the body of the request
  connect to the remote host and forward the request
  if (response is javascript) {
    return rewritten javascript as a response
  } else {
    return unmodified response
  }
}
```

Figure 3.7: Pseudocode of the HTTP Proxy.

3.4 Visualization

The visualization system follows the guidelines set out by Shneiderman [52]: "Overview first, zoom and filter, then details on demand" serves to guide the design of the MTWAVE visualization system. The system initially shows a high-level waterfall view that illustrates the concurrent HTTP requests that occur during a web page load. By clicking on one of the HTTP requests, the view changes to an inverted flame diagram inspired by Brendan Gregg.[5] This shows the temporal details of the execution of that particular request. Hovering over particular entries, the user is presented with the specific details of each function called and the execution timing associated with that invocation.

3.4.1 High-level View

The high-level waterfall view is a two-dimensional temporal view of the HTTP requests that constitute a page load. The horizontal axis represents the passage of time. Each entry along the vertical axis is a separate HTTP request, ordered by start time. Displaying each request on a separate line allows for both the inherent parallelism and the request blocking semantics of HTTP and HTML to be readily observed.

This visualization is critical for understanding the user-perceived performance; the top-left corner of the plot is the moment where the user clicked to load the page; the bottom-right corner is the moment where, from the user's point of view, the page is done loading. By inspecting this plot, we can immediately identify slow portions of the page load and begin to dig in. For example, in Figure 3.8 shows the high-level visualization. The horizontal axis represents the passage of time, and the vertical axis shows each HTTP request, ordered top-to-bottom by the start time. There are two obvious bottlenecks shown in this figure:

1. The very first request occupies nearly half of the timeline and blocks all other requests until it has finished executing. This is the request that is optimized in Section 4.3.1.

2. There is a blocking request about $\frac{1}{3}$ of the way down; this is a `<script>` tag that loads a Javascript file. The semantics of HTML and Javascript dictate that the browser must wait until that file has downloaded and executed before it may resume making additional HTTP requests.

There are cases where a page is never "done" loading, rather it will continue issuing dynamic requests for additional information. MT-WAVE will continue tracing the execution of requests until the user navigates to a new page. Loading the visualization before the user has navigated away or closed the page will display the data that has been collected so far.

[5] `http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html`

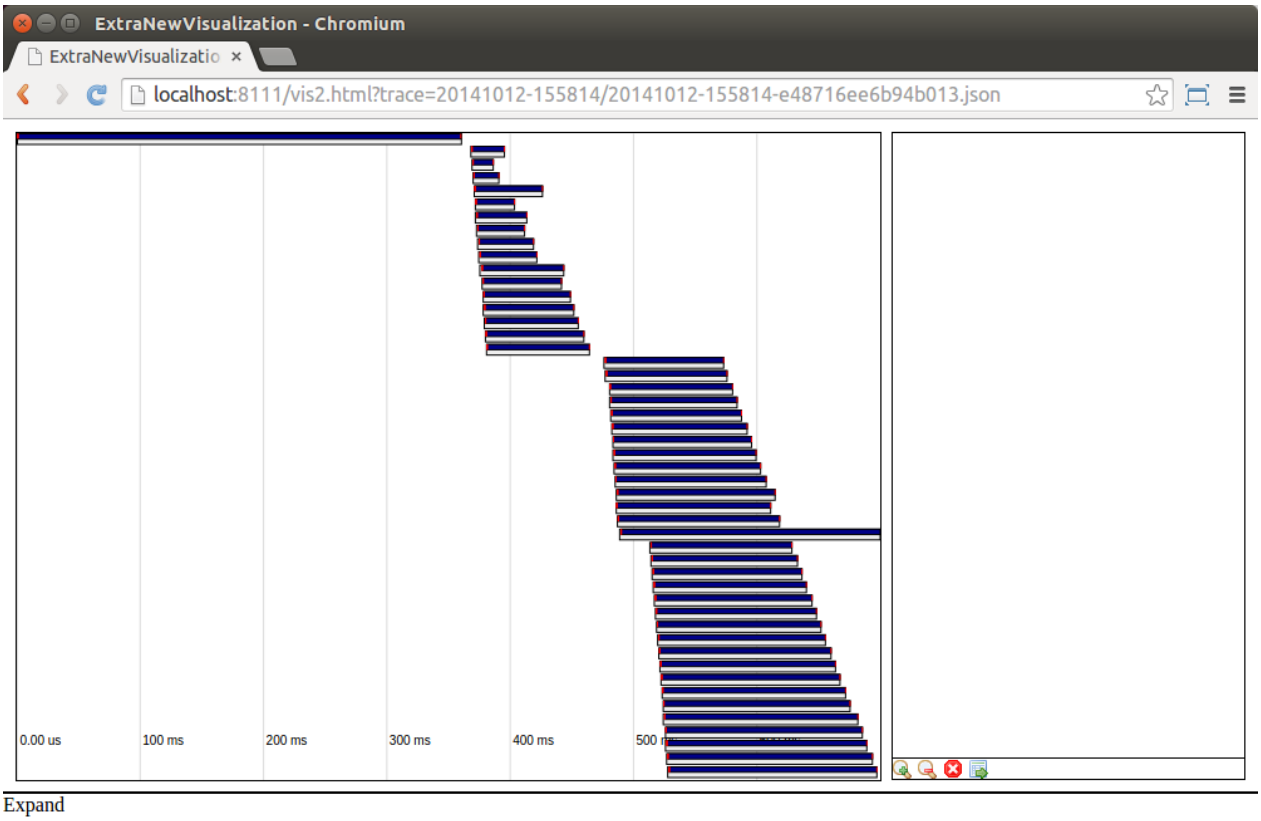


Figure 3.8: MT-WAVE Visualizer: High-level view.

3.4.2 Request Trace View / Flame Graph

This view represents the backend execution of a particular HTTP request. Similar to the waterfall view, the horizontal axis represents the progression of time. The vertical axis, however, represents the nesting of function calls within the call graph; if function `a` calls function `b`, `b` will appear underneath `a`. This structuring enables a fast visual investigation of the call graph to identify which function calls are the most likely candidates for optimization. Figure 3.9 shows an example request from the Magento case study (Section 4.3.1). The horizontal axis represents the passage of time, and the vertical axis shows the nested call graph of the chosen HTTP request.

An additional advantage of this structure is that one can quickly understand the execution of an unfamiliar software package. The experiments done in Chapter 4 were primarily done with no prior knowledge of the internals of the software packages; this big-picture view of the software was instrumental in rapidly understanding the structure of the packages.

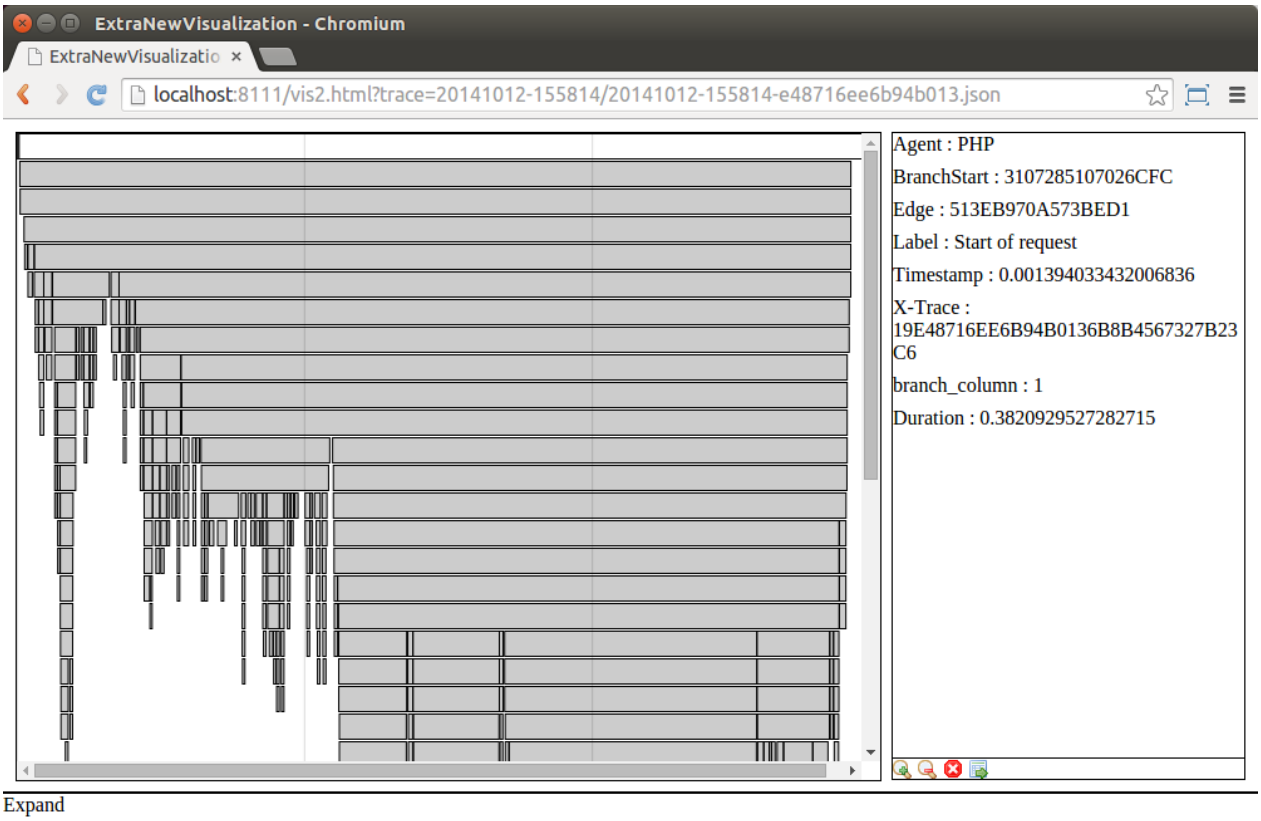


Figure 3.9: MTWAVE Visualizer: Request Trace view.

3.4.3 Specific Event Details

Moving the mouse over the various function calls in the Request Trace View results in a text-based representation of each function call being displayed on the right hand side; this view includes the name of the function, the source location of the code being executed, and the timestamp and execution time of the highlighted invocation. This view allows the analyst to rapidly find and inspect the executed code. Moving the mouse vertically through the Request Trace View moves through the call stack, providing the details of all of the caller/callee relationships in the execution. In Figure 3.10 the right-hand side shows the details of the function call that the mouse is hovering over on the left-hand side.

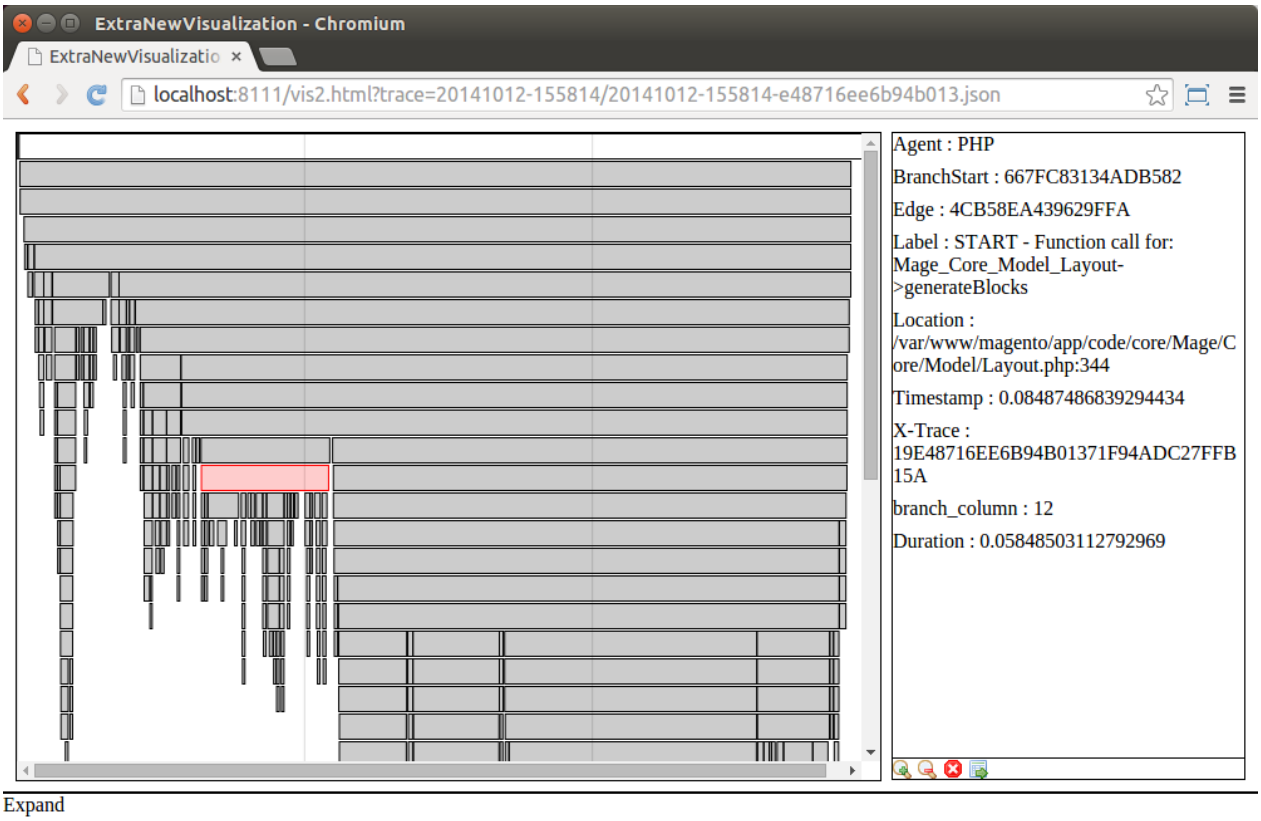


Figure 3.10: MTWAVE Visualizer: Specific Event details.

3.5 Repeatable Performance Evaluation Experiments

3.5.1 Scripted build configuration

The first step to having repeatable experiments is to fully automate the build process; every manual step in the build process is another point of misunderstanding. For these experiments, every step is automated and can be rebuilt from scratch with specific versions of software packages. Docker[6] is used to run each experiment in an isolated environment. While similar to virtualization, Docker relies on Linux Containers[7] for isolation.

Before each experiment is run, its Docker container is rebuilt from scratch to ensure that nothing in the environment has been accidentally modified (which could affect the result). To ensure that there are no accidental missed steps in this process, only a single command is required to build the environment; for example, "`cd docker/joomla && docker build -t joomla .`" is used to build the Joomla environment.

[6] `https://www.docker.com/`
[7] `https://linuxcontainers.org/`

3.5.2 Scripted experiments

Once repeatable software environments have been established, the experiments themselves must also be scripted. Any manual setup or configuration introduces another point of possible inconsistency between experiments.

Since MT-WAVE measures user-perceived latency, Selenium WebDriver[8] is used to automate user interactions through a live and fully functional web browser. With this library, Ruby scripts are written to perform every step of the experiment. This ensures that the MT-WAVE Firefox plug-in is configured consistently, and that each run of the experiment is properly labelled with a description, object, and source code revision identifier. The scripts are designed so that they will refuse to run if there is modified source code that has not yet been committed to the source control repository.

3.5.3 Data archival

Finally, once the experiments have been run, there must be a system in place to archive the raw data. This has three important consequences:

- There is a "gold standard" that others can compare their own experiments with
- Further data analysis is possible in the future
- Any questions about potentially anomalous data can be resolved without needing to re-run the experiment (and potentially observing different behaviour)

Out of the box, X-Trace does not support data archival and re-analysis in any kind of meaningful way. JSON support (see Section 3.2.3) provided for a simple plain-text but machine-readable data storage format. When one of the scripted experiments run, a unique tag (based on a timestamp) is assigned to all of the events that are generated from the browser. This unique tag is used to extract all of the X-Trace events generated by that experiment and save them into a JSON file. This file is written alongside the other data generated by the automated experiment script, and another experiment cannot be run until this data has been committed to the source control repository.

The visualization software is designed to parse this exported data file; this enables the data to be explored without needing a full experimental setup to be installed on the machine. In the original work, the MT-WAVE visualization software was tightly coupled to X-Trace, but this proved to be cumbersome for rapid development and exploration. Loose coupling through a well-defined data format (JSON) made for a (qualitatively) better development environment.

3.5.4 System Utilization Monitoring

To ensure that the measured system latencies are caused by the application execution performance and not by accidentally saturating computer subsystems, the host computer is monitored using Gregg's USE Methodology [20], using

[8] http://docs.seleniumhq.org/projects/webdriver/

the specific Linux commands outlined on Gregg's website.[9] Like all of the other captured data and metadata, another experiment cannot be run until this data has been committed to the source control repository.

3.6 Summary

MT-WAVE is a system built up of several pieces: a centralized event collection server based on X-Trace, a number of data collection modules for each tier of the application, and a visualization system designed to give varying levels of detail to facilitate understanding of how the application works and the sources of latency.

A Breadcrumb Tracing approach was chosen for cross-tier tracing. In this style of tracing, small pieces of metadata are passed between the tiers to allow for exact reconstruction of event causality. Other techniques exist, but these rely on heuristics or application-specific rules to be determined to establish causality between tiers; these other approaches are effective for systems where the source code is not available (Blackbox Tracing), but add unnecessary complication and risk of incorrect results for developers debugging their own applications.

The tracing modules in MT-WAVE follow a common pattern: intercept the request at the start of execution to parse the metadata, emit X-Trace events as execution proceeds, and finally emit an event at the end of execution. The BranchId extension for X-Trace ensures that both the function invocation and return events can be reconstructed into a precise execution trace during visualization. Modules have been written to instrument Firefox, Python, Ruby, Javascript, and PHP.

The visualization system follows an established pattern for human-computer interaction: "Overview first, zoom and filter, then details-on-demand". The user is presented first with an overview showing all of the requests and durations, which clearly shows which requests are adding the most latency to the overall action. Clicking on a particular request zooms in to a trace view that shows a "flame graph" that simultaneously shows the call graph and function call latency. Moving through this graph shows the precise details about which functions were called from which files, including the source line numbers.

Finally, a system is set up to script data collection. The developer can specify which URLs are to be retrieved and how many times each action should be done. This automated infrastructure ensures that experiments can be repeated without accidentally changing aspects of the experiment that could influence the underlying system performance. The output from these scripts is processed to provide concise summary views of the system performance that make it easy to compare before-and-after performance of the system while potential latency improvement changes are taking place.

[9] http://www.brendangregg.com/USEmethod/use-linux.html

CHAPTER 4

CASE STUDIES & EVALUATION

4.1 Introduction

This chapter describes a series of case studies used to evaluate the performance and effectiveness of MT-WAVE. To start, we establish the baseline performance of several web applications with and without MT-WAVE to determine how much overhead MT-WAVE adds in different situations. Then, MT-WAVE is used to improve performance in these applications, with a description of what was changed and how MT-WAVE was used to improve system performance.

4.1.1 Analysis Techniques

For establishing baseline performance, a number of samples are taken using a zero-overhead technique (window.performance.timing, Section 2.7.5) and the collected timing is summarized using Student's T-Test to establish the mean and 95% confidence interval.

To compare before and after performance, Welch's Unpaired Two Sample T-Test is used. This establishes a 95% confidence interval on the difference of the sample means; if 0 is not within this confidence interval, there was a statistically significant performance difference between the two experiments.

Sample sizes are chosen as a trade-off between accuracy and measurement time. While iterating on a potential performance improvement, a quick determination of whether or not a particular change has a positive impact is desirable; a smaller sample size is chosen to get that feedback without having to wait too long. Once a set of fixes have been applied, the performance improvement is verified by using a larger sample size. In these experiments, a sample size of 100 is typically chosen for the "rapid feedback" case; this is large enough to ensure that the T-Test will not be impacted by potential non-Gaussian distribution within the sample. For verification, a sample size of 1,000 is chosen; this was originally chosen using a "rule of thumb" for large sample sizes. As the data was collected, it is clear that the before-and-after means were sufficiently large (and the confidence intervals sufficiently narrow) that there was little doubt that large performance changes had occurred.

4.1.2 Experiment Selection and Planning

Since MT-WAVE is designed to debug performance problems in production-ready web applications, the large number of readily-available open source applications is leveraged. Each of the case studies follows a general template:

1. Identify a popular open-source application written in a programming language that we have instrumentation prepared for.

2. Build a repeatable environment that hosts the application and MT-WAVE instrumentation plug-in.

3. Establish baseline performance without MT-WAVE.

4. Compare baseline performance of the application with MT-WAVE against the initial baseline performance (determine MT-WAVE overhead)

5. Look at the MT-WAVE traces to identify a potential candidate for performance optimization. Generally, the best potential candidate for optimization is one of the longest-running function calls within the application; by fixing this function, we will have the largest overall improvement.

6. Make a modification to the candidate function that attempts to improve the performance. Measure to determine whether or not the modification made a significant improvement.

7. Repeat until performance is acceptable. After each modification, the most expensive function call is likely to change; we continue by targeting whichever function call is currently the most expensive.

8. Once performance has sufficiently improved, a large number of samples (without MT-WAVE) to verify that a real world improvement to the software has been achieved, instead of just influencing the run-time of the MT-WAVE instrumentation.

4.1.3 Summary of Analyzed Applications

Joomla[1] is a popular PHP-based Content Management System (CMS). It was chosen primarily due to its popularity and ease of installation; as a popular CMS, it serves as a good example of the type of PHP system that MT-WAVE would be used to debug.

Magento[2] is a PHP-based e-commerce platform. Like Joomla, it was chosen because it is a popular open-source PHP application. After doing a default install and configuration (using the recommended Magento sample database), the software subjectively felt slow, which made it an excellent target for optimization. The optimization process demonstrates the difficulty in determining which code to modify to improve performance; the first two experiments resulted in performance losses, but MT-WAVE provided direct insight into the software behaviour and how the changes were affecting the system.

Wordpress[3] is an extremely popular content-management system, used primarily for hosting blogs. After installation, page loads subjectively felt "almost instantaneous" and it seemed unlikely that significant improvements could be made. Looking at the MT-WAVE traces confirmed this suspicion; the majority of the user-visible latency was the result of the code being reloaded from disk every time (it took 2.88× longer to load the code than it took to run it).

[1] http://www.joomla.org/about-joomla.html
[2] http://magento.com
[3] http://wordpress.com

Social Spiral[4] is a commercial application written in Ruby on Rails, with a large JavaScript-based front-end. Although it was not suffering from any known performance deficiencies, it is a sufficiently large codebase to use as a demonstration for MT-WAVE. Running MT-WAVE against the application resulted in several opportunities for optimization, as well as an excellent illustration of some of the difficulties and unintuitive results that can be encountered when working on web applications written in interpreted languages.

4.2 MT-WAVE Overhead Evaluation

4.2.1 Summary

The objective with these experiments is to determine the overhead imposed by MT-WAVE when profiling several different software packages. Browser-perceived performance in all cases is determined using `window.performance.timing`; data is collected using a script that records this data for 1,000 page loads with and without MT-WAVE plug-ins active.

These experiments establish an average overhead of 237 ms, which is low enough that the tool is usable. Despite the overhead, the tool was used to effectively find and solve several performance problems in Section 4.3. Table 4.1 summarizes the various overhead experiments.

Table 4.1: Summary of overhead experiments.

Package	Experiment	Mean (ms)	Change	
			Lower (ms)	Upper (ms)
Joomla	J1	338.5	Baseline	
Joomla	J4	532.5	+176.2	+211.8
Magento	M1	659.1	Baseline	
Magento	M2	1397.6	+ 714.9	+761.9
Wordpress	W1	230.7	Baseline	
Wordpress	W2	539.8	+297.6	+320.7
Satchmo	Sa1	60.6	Baseline	
Satchmo	Sa2	122.8	+50.1	+74.3
Trac	T1	205.8	Baseline	
Trac	T2	282.4	+67.2	+86.1
Social Spiral	So1	234.6	Baseline	
Social Spiral	So2	276.4	+30.4	+53.4
Mean			+237.0	

4.2.2 Joomla

These experiments start by measuring the baseline performance of Joomla (without MT-WAVE installed). MT-WAVE instrumentation is then added and system performance is measured again, to determine the overhead caused by the introduction of MT-WAVE into the system.

[4] `http://www.socialspiral.com`

To measure the baseline performance of Joomla, a simple Selenium WebDriver script is set up to load the home page of a Joomla installation and record the browser-perceived load time. Since the objective is to measure the page-load performance without any special plug-ins, the least invasive method for determining user-perceived latency is used: window.performance.timing. This technique gives us no insight into the source of latency; it simply reports the browser-perceived page-load time.

In the context of PHP-based experiments, there are two potential sources of increased overhead: the in-browser MT-WAVE plug-in and the PHP tracing plug-in. To evaluate the impacts of these, four experiments were run:

- J1: No plug-ins installed at all. This establishes the baseline performance.
- J2: Firefox plug-in installed, PHP plug-in not installed. This evaluates the overhead due to the in-browser capture mechanism.
- J3: Firefox plug-in not installed, PHP plug-in installed. This evaluates the bare overhead of the PHP plug-in when tracing is inactive.
- J4: Firefox and PHP plug-ins installed. This evaluates the total overhead when full-stack tracing is active.

All experiments were run with 1,000 samples. The collected data is summarized in Table 4.2 and plotted in Figure 4.1.

One interesting result is that the PHP plug-in appears to have no performance impact when it is the only plug-in installed; this is because the only processing the PHP plug-in will do without the Firefox plug-in is to look for, and not find, the MT-WAVE HTTP header. There will be a small impact (there must be, as additional code is executing), but that impact is so small that it is lost in the measurement noise.

Otherwise, the results generally match our natural intuition: the system performance slows down when instrumentation is added, and for a given condition the system performance remains stable over time. The interesting part of Figure 4.1 is that it does not show any unintuitive result. During the initial run (see Section 4.4) this was not the case: system performance degraded over time due to a bug in MT-WAVE.

Table 4.2: User-perceived performance of Joomla, with and without MT-WAVE tracing plug-ins enabled.

Experiment	Mean	Change - Lower CI	Change - Upper CI	Conclusion
J1	338.5 ms	—	—	—
J2	467.6 ms	111.9 ms	146.1 ms	worse
J3	337.0 ms	-16.3 ms	13.3 ms	no change
J4	532.5 ms	176.2 ms	211.8 ms	worse

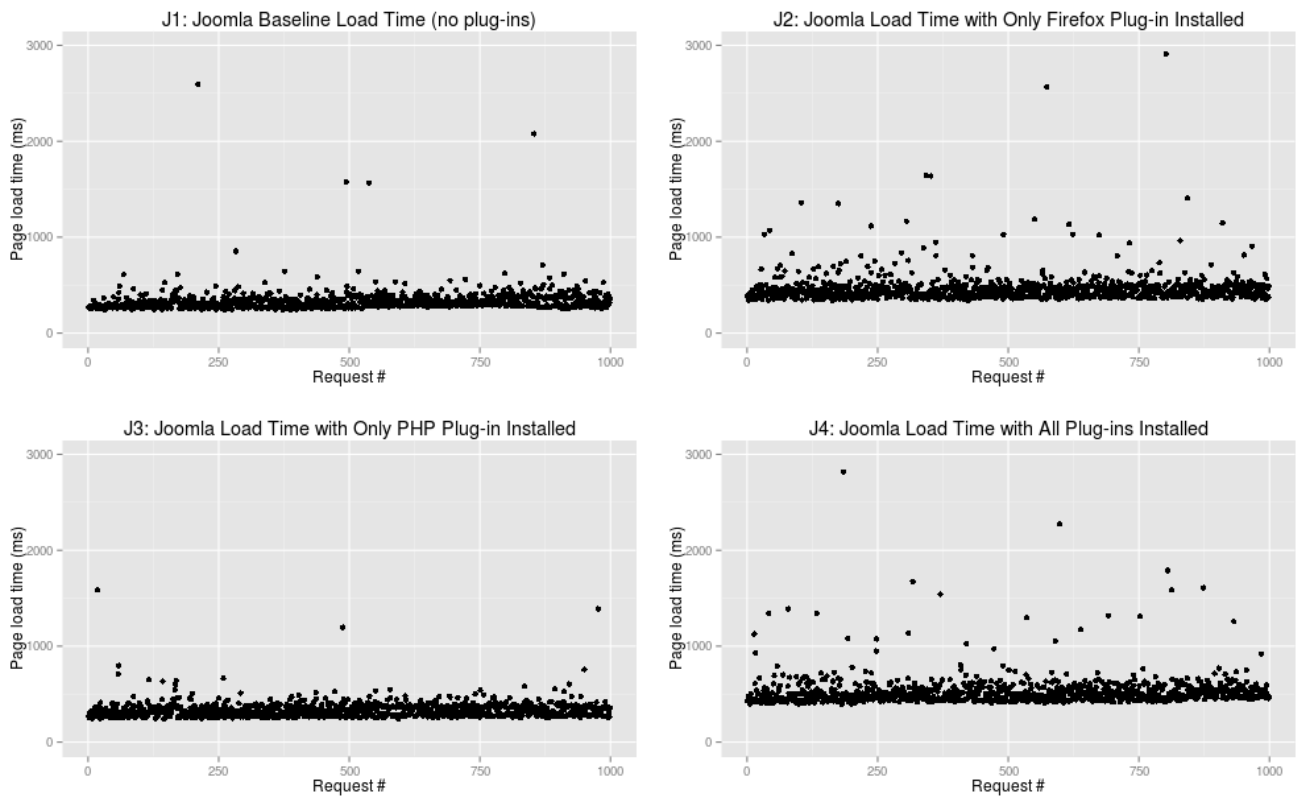


Figure 4.1: Joomla performance with different MT-WAVE tracing plugins enabled.

4.2.3 Magento

Similar to the Joomla experiments, the initial measurements were done against a stock Magento install with and without MT-WAVE installed. This is used to determine both the baseline performance of the system, and the MT-WAVE-imposed overhead.

As the Joomla overhead experiment clearly demonstrated that the MT-WAVE overhead only happens when all tracing plug-ins are enabled, only two overhead experiments were run against Magento:

- M1: Baseline Magento without MT-WAVE and without any modifications.
- M2: Magento with MT-WAVE, but without any modifications.

Figure 4.2 shows the response times with and without MT-WAVE, and the data is summarized in Table 4.3. In this experiment, running with MT-WAVE takes 2.1× longer (1397 ms compared to the baseline 659 ms).

Table 4.3: Baseline user-perceived performance of Magento, with and without MT-WAVE running.

Experiment	Mean	Change	
		Lower	Upper
M1 (Baseline)	659.1	–	–
M2 (MT-WAVE)	1397.6	+714.9	+761.9

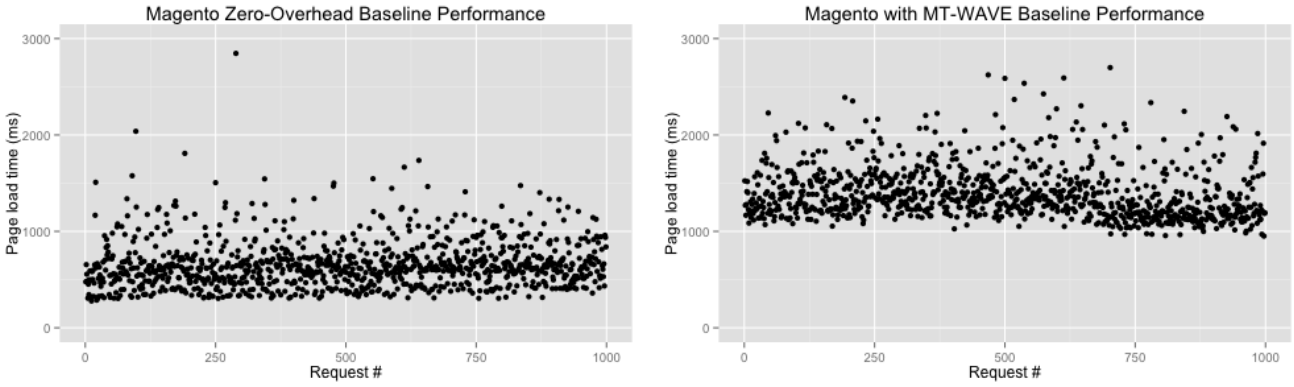


Figure 4.2: Baseline user-perceived performance of Magento, with and without MT-WAVE running.

4.2.4 Wordpress Overhead

The Wordpress overhead experiments closely follow the design of the Magento experiments: use zero-overhead profiling to establish a baseline (W1), and run the same request again using full tracing (W2). Table 4.4 shows the results of these experiments.

Table 4.4: Baseline user-perceived performance of Wordpress, with and without MT-WAVE running.

Experiment	Mean	Change	
		Lower	Upper
W1 (Baseline)	230.7	–	–
W2 (MT-WAVE)	539.8	+297.6	+320.7

4.2.5 Satchmo and Trac Overhead

The Satchmo and Trac experiments are grouped together because of their similarity: both are Python applications that had no obvious performance problems identified in the initial tracing. For these experiments, two experiments were run. First, with zero-overhead tracing (Sa1, T1), and then with MT-WAVE enabled (Sa2, T2). In this case, the Python tracing plug-in does not trace the complete execution of the application, but rather it only records user-specified events (see Section 3.3.1). This means that the measured overheads are the minimum overhead for tracing; the only events

being recorded are the start and end of execution. This does account for all of the setup and teardown required for MT-WAVE tracing: parsing the X-Trace metadata, setting up a UDP connection to the X-Trace server, and generating the internal data structures required for tracing. Table 4.5 summarizes the results.

Table 4.5: Baseline user-perceived performance of Satchmo and Trac, with and without MT-WAVE running.

Experiment	Mean	Change	
		Lower	Upper
Sa1 (Baseline Satchmo)	60.6	–	–
Sa2 (MT-WAVE Satchmo)	122.8	+50.1	+74.3
T1 (Baseline Trac)	205.8	–	–
T2 (MT-WAVE Trac)	282.4	+67.2	+86.1

4.2.6 Social Spiral Overhead

The Social Spiral overhead experiments were performed against the performance-enhanced version. The experiment measures the time required for the initial request (`/retail/`) to execute, in order to get a better measure of the tracing overhead incurred on a single request; for an application that makes multiple backend requests to the application, this 30 ms to 50 ms overhead will be incurred for each request. Table 4.6 summarizes the results.

Table 4.6: Baseline user-perceived performance of Social Spiral, with and without MT-WAVE running.

Experiment	Mean	Change	
		Lower	Upper
So1 (Baseline)	234.6	–	–
So2 (MT-WAVE)	276.4	+30.4	+53.4

4.3 Performance Improvements

4.3.1 Magento - Index Page Latency

The Magento index page (first page loaded when a user visits a Magento site) follows a standard request model. First, the HTML for the page is retrieved. The HTML is parsed, and the browser generates a list of resources needed to fully display the page: JavaScript, stylesheets, images, etc. As soon as this list is ready, the browser begins requesting these additional resources. Figure 4.3 shows an annotate diagram from MT-WAVE that highlights the different types of resources retrieved.

Figure 4.3 makes it clear that the initial request for Magento dominated the total page load time, and will be the target of this experiment. There are a few other observations that can be made here as well.

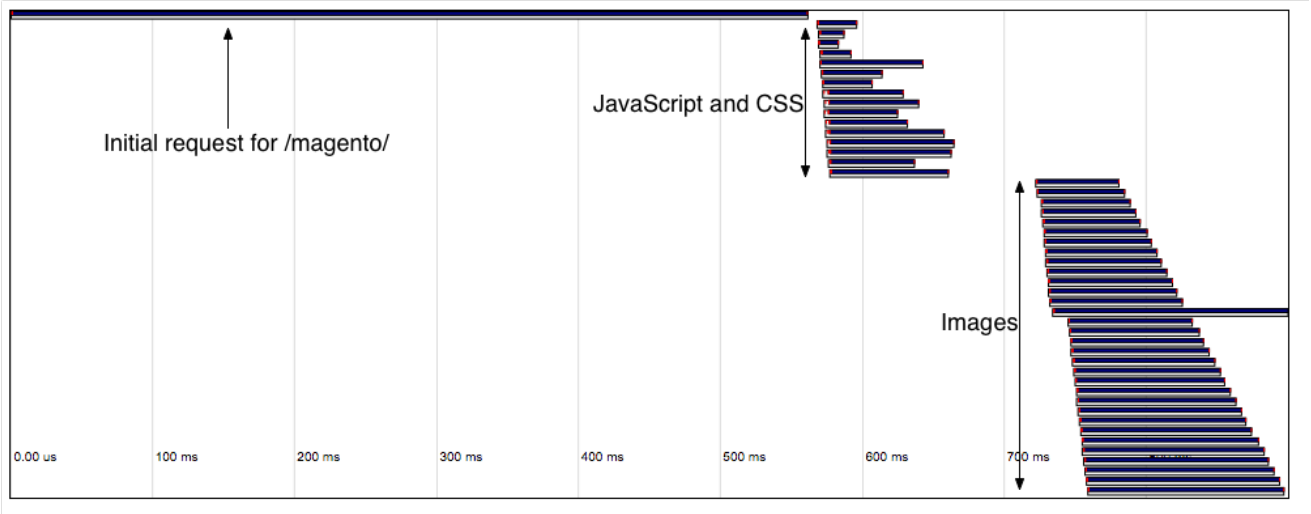


Figure 4.3: Outline of the resource requests during a Magento page load.

First, the images were not requested until all of the CSS and JavaScript had been downloaded and parsed. This is because JavaScript is executed immediately after it has been parsed, and this code can modify the HTML document. In other words, parsing JavaScript is blocking, and the browser must wait until the JavaScript code has finished executing before it can continue retrieving images and other resources. A common optimization is to add the JavaScript `<script>` tags to the end of the HTML document to mitigate this behaviour; JavaScript is still blocking, but the browser has already queued up a large list of resources that will continue downloading while the JavaScript code is executing.

A second observation is that there is a large number of CSS, JavaScript, and image requests that are made. The images are understandable; Magento is an e-commerce platform, and the index page shows images of popular products. The JavaScript and CSS files, though, could be reduced down to a single `.js` file and a single `.css` file, using a tool called a Minifier.[5] Looking at this trace, though, it's clear that despite these two problems, the largest source of latency is in the code that generates the initial HTML.

While exploring the MT-WAVE traces of the Magento index HTML generation, it was observed that only a small portion of the server-side processing was spent generating the HTML that was to be displayed; a significant portion of the processing time was spent logging information regarding the current user's session. An iterative approach was unintentionally taken to reduce this latency; the first two attempts at reducing it ended up causing an unexpected increase in latency, which was readily apparent in the MT-WAVE traces. Figure 4.4 shows the portion of the request dedicated to generating HTML; the highlighted function performs all of the processing required to render the Magento home page. This function is only responsible for 277ms, of the total 521ms required to return the page to the browser.

The Magento codebase is structured as a series of "Controllers", each of which is responsible for generating the HTML for a specific URL. Inspecting the `IndexController`, there was logic that generated the index HTML, and there was logic that recorded the last URL that the user had visited. Experiment "M3" was to simply comment out the

[5] `http://yui.github.io/yuicompressor/`

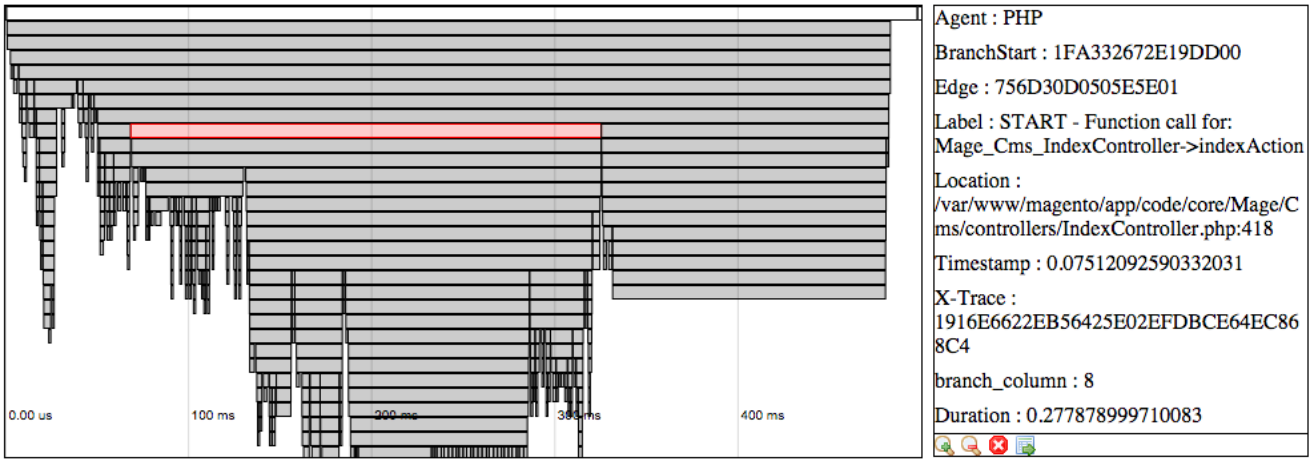


Figure 4.4: Time spent generating HTML in baseline Magento.

Table 4.7: User-perceived performance of Magento: 95% confidence intervals comparing the change of the mean performance of each modification against the initial baseline performance.

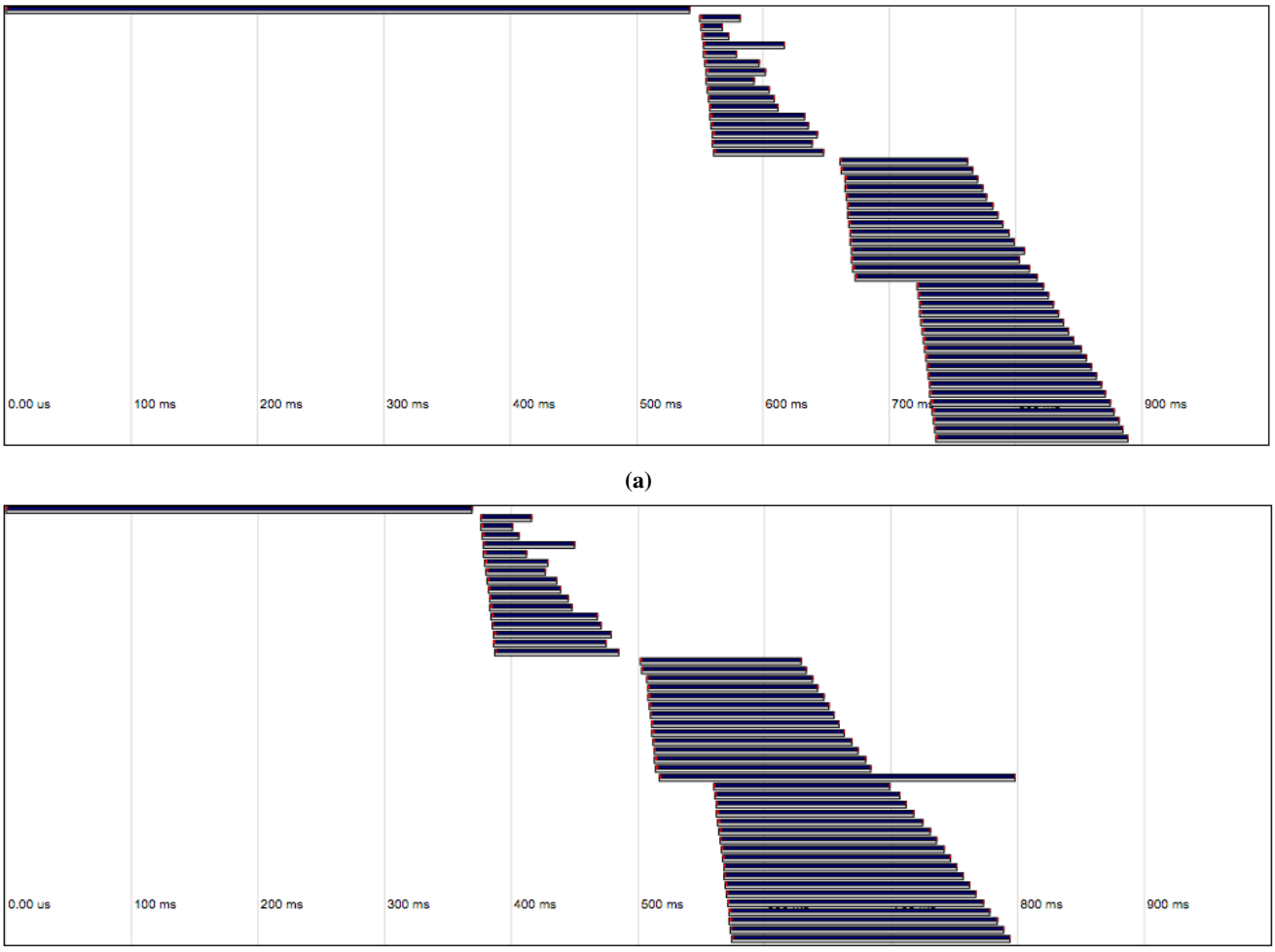
Experiment	Mean (ms)	Lower CI (ms)	Upper CI (ms)	Conclusion
Baseline	1256	–	–	–
M3	1354	+38.33	+158.32	worse
M4	1481	+158.47	+292.27	worse
M5	1077	-240.51	-116.10	better

URL recording logic.

Commenting out the URL recording logic had an unintended side-effect. It appears that the recording logic set a flag indicating that other session-tracking code did not have to run; as such, the other tracking code now ran all the time and caused a slight increase in latency. Experiment "M4" disabled this post-request tracking code by removing a few lines of XML from the logging config file.

This, too, had an unintended side-effect. Now, an even more expensive session-tracking function was running at the beginning of each request (instead of at the end of the request). In Experiment "M5", this code was also disabled by removing a few lines of XML. With that functionality removed, latency finally began to improve. Figure 4.5 shows the difference in user-perceived latency between the baseline measurement and the results of M5.

To compare the system performance between the different iterations, the distributions of user-perceived latency are compared using the Welch Two Sample t-test. In all cases, $P \ll 0.05$, indicating that there is a statistically significant difference in means. The test calculates 95% confidence intervals on the difference in means between the distributions, summarized in Table 4.7.



(a)

(b)

Figure 4.5: Comparison between the baseline Magento performance (a) and the final improved performance (b).

4.3.2 Wordpress - Adding Bytecode Caching

Comparing the Wordpress and Magento index pages demonstrates the performance benefits that can be achieved by reducing the number of requests for JavaScript and CSS. Figure 4.6 shows the comparatively simple set of requests made to load Wordpress: the initial HTML, a non-standard font needed to display the page, two CSS files, and three JavaScript files.

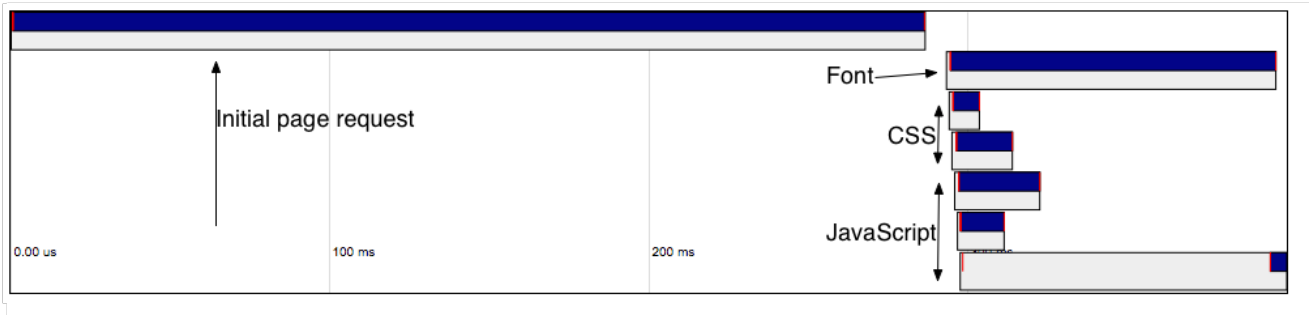


Figure 4.6: Outline of the resource requests during a Wordpress page load.

Without any changes, Wordpress is relatively fast when compared to the other packages investigated; mean user-perceived performance is 542.8 ms, with a 95% confidence interval of $[512.5, 573.3]$. From the user's point of view, this performs very well. For scalability reasons, though, the site owner may want to improve this further.

Inspecting the MTWAVE traces show that the majority of the user-perceived latency happens in the PHP request itself. Zooming into the PHP request, there are two main scripts being called: `wp-load.php` and `index.php`. These are, respectively, loading the Wordpress framework and preparing the index page for HTML rendering. In the sample request used, loading takes 179 ms and rendering takes 62 ms. The code takes $2.88\times$ longer to load than it does to execute. This suggests that speeding up the loading process has the most potential for speed improvements.

One approach to improving PHP load times is to implement a bytecode cache. APC[6] is the canonical PHP bytecode cache; typically, this does not require any code modification at all, just installation of APC and configuration. Adding this to the Wordpress installation provided a reduction in latency: mean 474.43 ms, 95% CI on the change $[-111.0, -25.9]$ with $P \cong 0.001$. Figure 4.7 shows the before and after performance of the system.

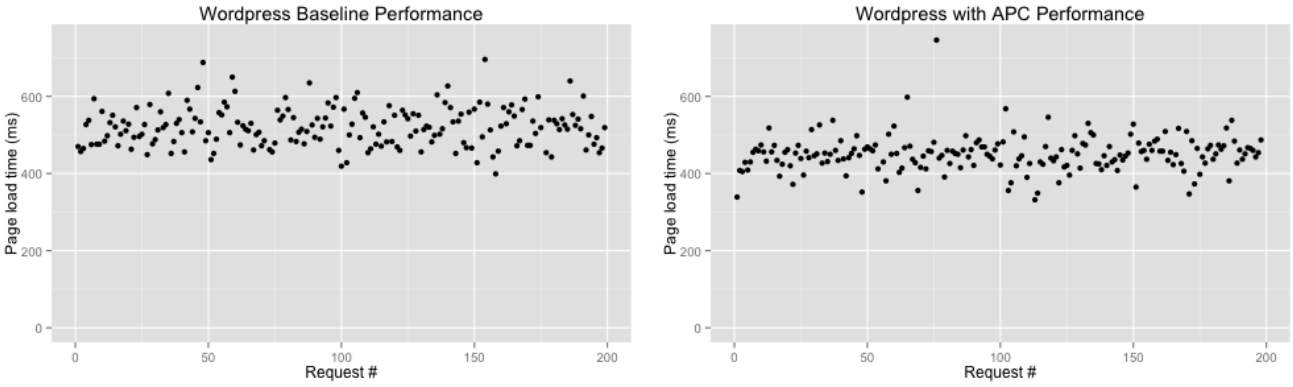


Figure 4.7: User-perceived Wordpress performance improvement when adding APC for bytecode caching.

Looking directly at the PHP performance, APC reduced the execution time of the Wordpress code from $\mu = 315$ ms to $\mu = 277$ ms with 95% CI $[-49.3, -27.5]$ at $P \cong 2.7e-11$, corresponding to a reduction of 12%.

[6] `http://php.net/manual/en/book.apc.php`

4.3.3 Social Spiral - Interpreter Lock & Caching

Social Spiral was significantly more dynamic than either Magento or Wordpress; the HTML was loaded once, and the browser updated dynamically as the user navigated through the application. The browser trace in Figure 4.8 shows this behaviour. The browser first loaded a small HTML file, two fonts, three JavaScript files, and a CSS file. After the initial set of files was loaded, the page was ready for the user to interact with; however, the JavaScript code continues to request content that will be used to fill in the rest of the page. This includes details about the current user and content they've stored, and a list of products. After the list of products was loaded, a series of image requests took place so that they could be displayed in the page.

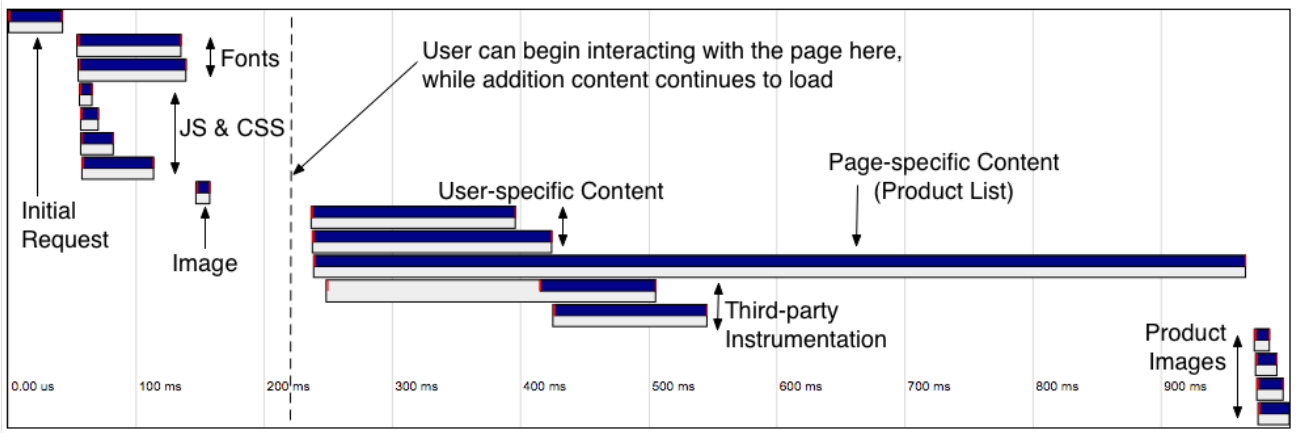


Figure 4.8: Outline of the resource requests during a Social Spiral initial page load.

4.3.3.1 Database Contention

Since there was no known performance problem, the first experiment with Social Spiral was to simply run it with MT-WAVE tracing enabled and inspect the execution traces. An obvious flaw stood out: a significant part of the execution time was spent waiting to acquire a database connection.

Rails applications can be either multi-threaded or multi-process, each with advantages and disadvantages. Both cases optimize request serving latency by creating a pool of serving threads/processes at start up to minimize delay when a request in received. In the multi-threaded case, the request-serving threads share a database connection pool; instead of each request handler making a new connection to the database, persistent connections are stored in the thread pool and retrieved on-demand. Due to a misconfiguration, Social Spiral had 16 worker threads, but only 5 database connections in the connection pool.

This application is a "white label" application, where customers can brand the application using their own logos, colour schemes, etc. and sell it to their own customers. Based on the domain name in the request, the presented application is modified to have the correct visual presentation for the customer. This information is stored as a database record. As a result, every request will need to make at least one database query; this means that the thread pool size and the database connection pool size should be equal to ensure that no thread ever has to wait to acquire a database

connection. Figure 4.9 shows the portion of the request waiting on the database connection pool to have an available connection. In this particular request, 759 ms was spent waiting of a total 1193 ms spent servicing the request.

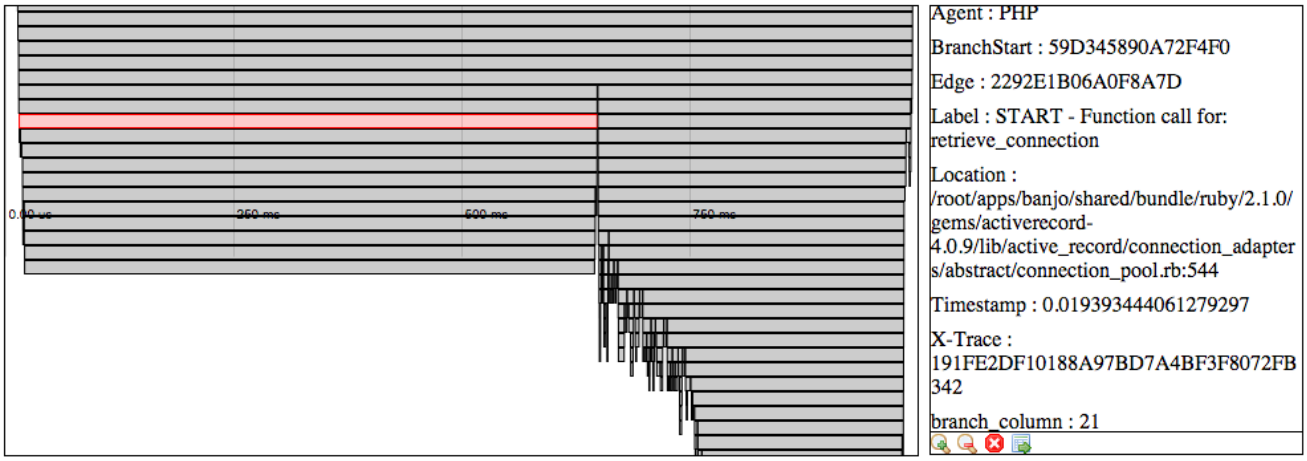


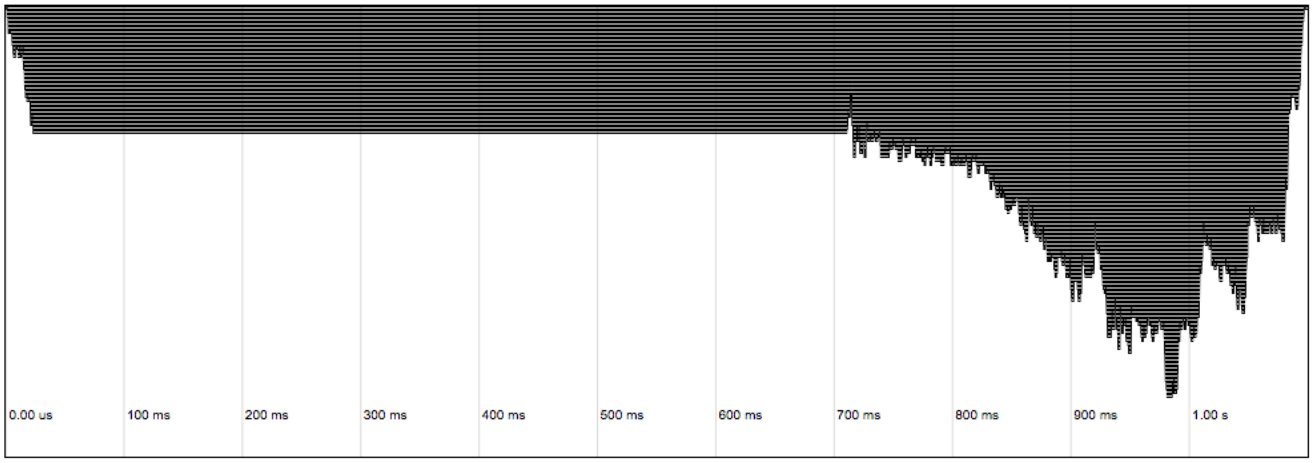
Figure 4.9: Social Spiral request handlers waiting to acquire a database connection.

Reconfiguring the application such that the thread pool and database connection pools were both set to 16 causes an unintuitive result. The request traces show that threads are no longer blocking to wait for a free database connection from the pool, but the system performance has remained unchanged! The per-function execution times in the reconfigured system have slowed down, resulting in nearly identical request processing times. Figure 4.10 shows a zoomed out view of the requests which shows that they have identical shapes, corresponding to identical call graphs (but stretched in time).

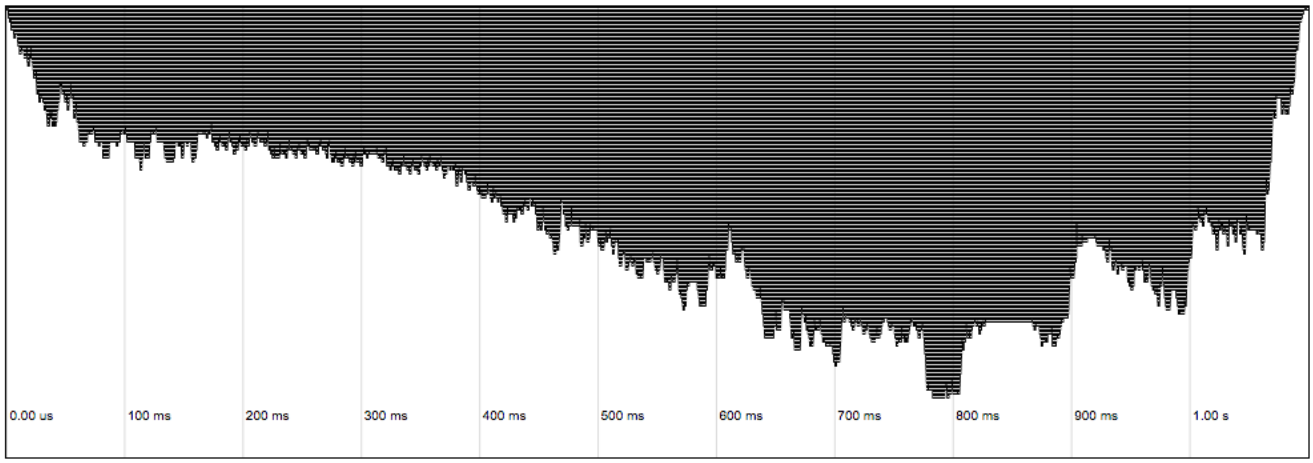
This is likely caused by the Global VM Lock: the standard Ruby 2.0 interpreter uses a single lock that a thread must hold to execute Ruby code [39]. When multiple threads are executing Ruby code, the lock is periodically released so that other threads can execute; the lock is also released when a thread makes a call into a C library or waits on a monitor, condition variable, or lock. The net result is that, even in a multi-core environment, the threads run cooperatively and compete for a single CPU core. In this case, when the thread waits to acquire a database connection (and releases the VM lock while waiting), there is less CPU contention and the other threads execute more quickly. The Global VM lock is an implementation detail of the Ruby interpreter that is not directly exposed to Ruby programs that are executing. This means that our Ruby-based instrumentation can not positively identify the VM lock as the source of latency; to do so would require a more elaborate module written in C.

A potential solution to this problem is to switch to a multi-process model instead of a multi-thread model for the application server. This is a major change and, because of the commercial nature of the application, has not been investigated further; the current server having been reliable for a long time, and this change has potential to introduce instability.

In a multi-threaded Ruby application, the loaded modules are shared between all of the threads. In a multi-process (fork) system, these in-memory data structures start out shared as well, but any modification to the structures causes the memory to be copied before the modification can take place (due to copy-on-write semantics in the Linux kernel [2]).



(a)



(b)

Figure 4.10: Comparison between Social Spiral with an inadequate database connection pool (a) and a properly-sized connection pool (b).

Intuitively, the loaded code should never change, but in practice, the Mark & Sweep garbage collector modifies this data in the Mark phase of collection. This was a significant problem in Ruby 1.9.3, but may have been successfully addressed in Ruby 2.0.[7] Before changing the application server over to a multi-process model, the extent of the increased memory usage must be evaluated.

To further complicate the issue, this problem disappears on a lightly loaded system; when there is no thread contention for the CPU or database pool, things execute quickly and there is no obvious lock contention in the execution traces. Any benchmarking to compare between multi-threaded and multi-process application servers needs to happen while the servers are under load.

Further evaluation of the performance and resource trade-offs between thread- and process-based Ruby performance is outside of the scope of this work. It is, however, worth further investigation in the future. Existing developer

[7] http://patshaughnessy.net/2012/3/23/why-you-should-be-excited-about-garbage-collection-in-ruby-2-0

guidance is typically simply prescriptive,[8] feature-based instead of performance-based,[9] or based on synthetic measurements.[10] Measuring and publishing the results using a live application with both real and synthetic loads would be beneficial to the Ruby on Rails community.

4.3.3.2 Multiple Requests

As is typical in front-end-centric applications, this application serves the initial HTTP request quickly, and then the front-end application makes a series of follow-up requests. Looking at the execution traces, it's clear that the `/retail/walls.json` request takes a large fraction of the page load time. Figure 4.11 shows the waterfall view of the page and the detailed view showing the time spent doing database lookups. A significant portion of this request is spent retrieving rarely-changed data from the database.

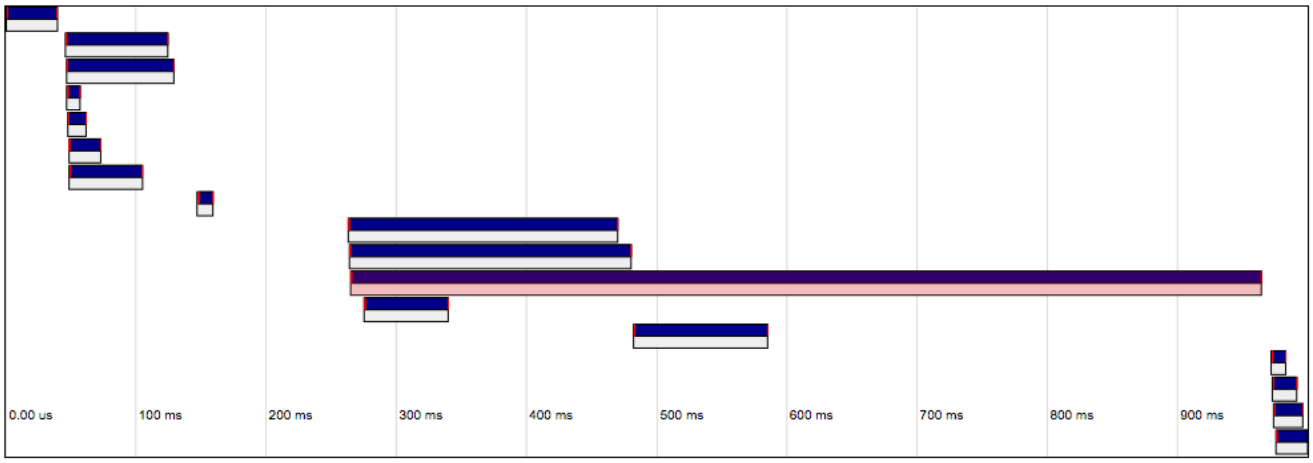
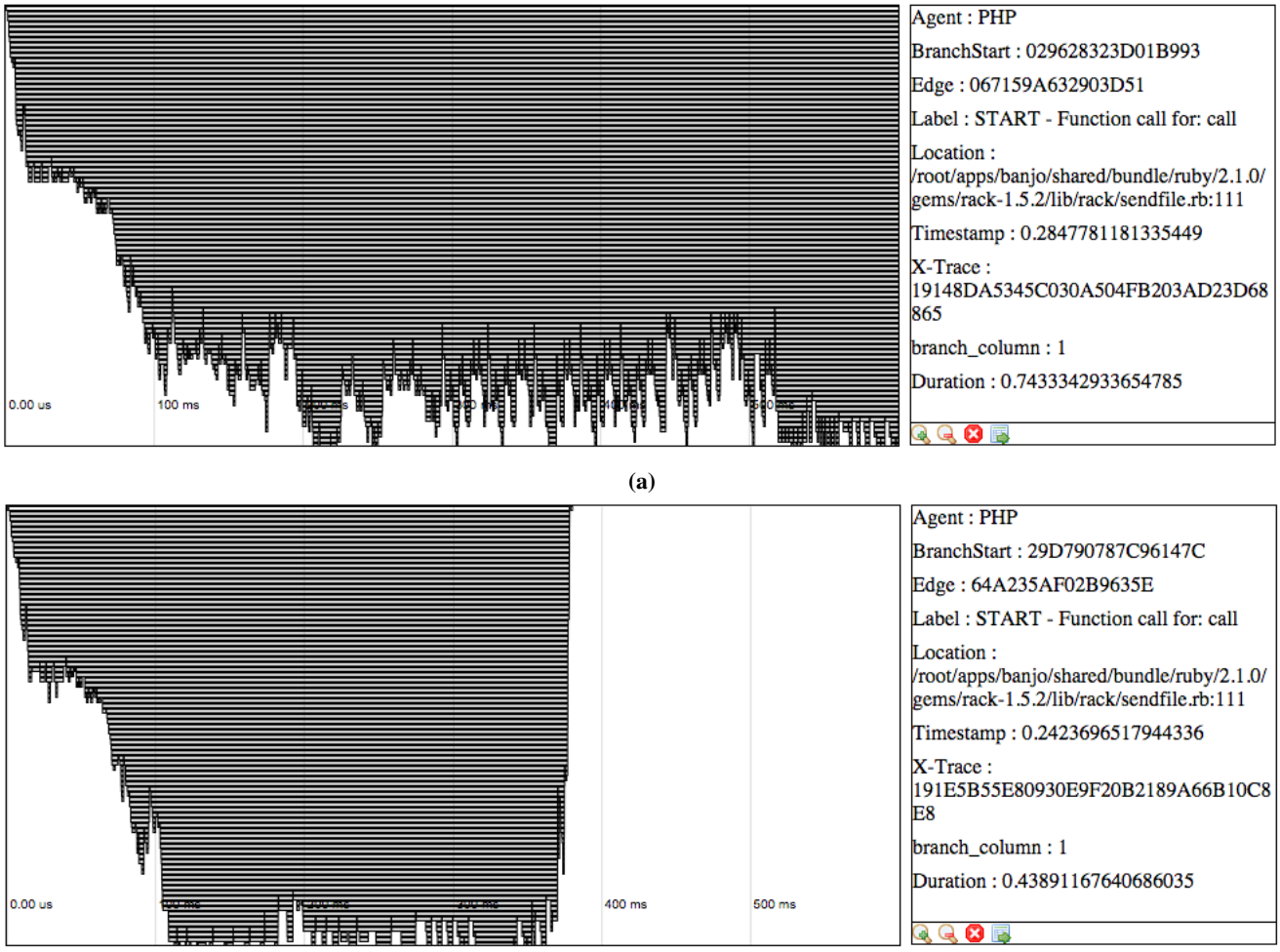


Figure 4.11: The highlighted trace is the long-running request for `/retail/walls.json`.

The content of `/retail/walls.json` changes infrequently and is retrieved frequently, making it an ideal candidate for server-side caching. Rails provides post-save hooks that are executed after a database record is modified, which allows us to do caching with an indefinite timeout; the cached records are invalidated when the underlying database records are modified or deleted. Caching reduces the mean request time for `/retail/walls.json` from 751 ms to 419 ms (confidence interval on change $[-340\,\text{ms}, -324\,\text{ms}]$), and reduces the overall page load time from 1082 ms to 719 ms (confidence interval on change $[-408\,\text{ms}, -317\,\text{ms}]$). Figure 4.12 compares the before and after execution traces.

[8] `https://devcenter.heroku.com/changelog-items/594`
[9] `https://www.engineyard.com/articles/rails-server`
[10] `http://www.akitaonrails.com/2014/10/19/the-new-kid-on-the-block-for-ruby-servers-raptor`



(a)

(b)

Figure 4.12: Comparison of the performance of `/retail/walls.json` between the uncached implementation (a) and the caching implementation (b).

4.4 Debugging MT-WAVE Firefox Plug-in Performance

This experiment was unplanned, but demonstrates the utility of MT-WAVE for determining which tier of the application stack is likely to be the source of performance problems. During the initial automated run of the Tracing Overhead data collection (Section 4.2), the collected overhead data showed that the system would get slower as the experiment progressed; i.e. the collected overhead measurements had a positive slope. In Figure 4.13, the slope is visually quite obvious in the cases when the Firefox plug-in is enabled.

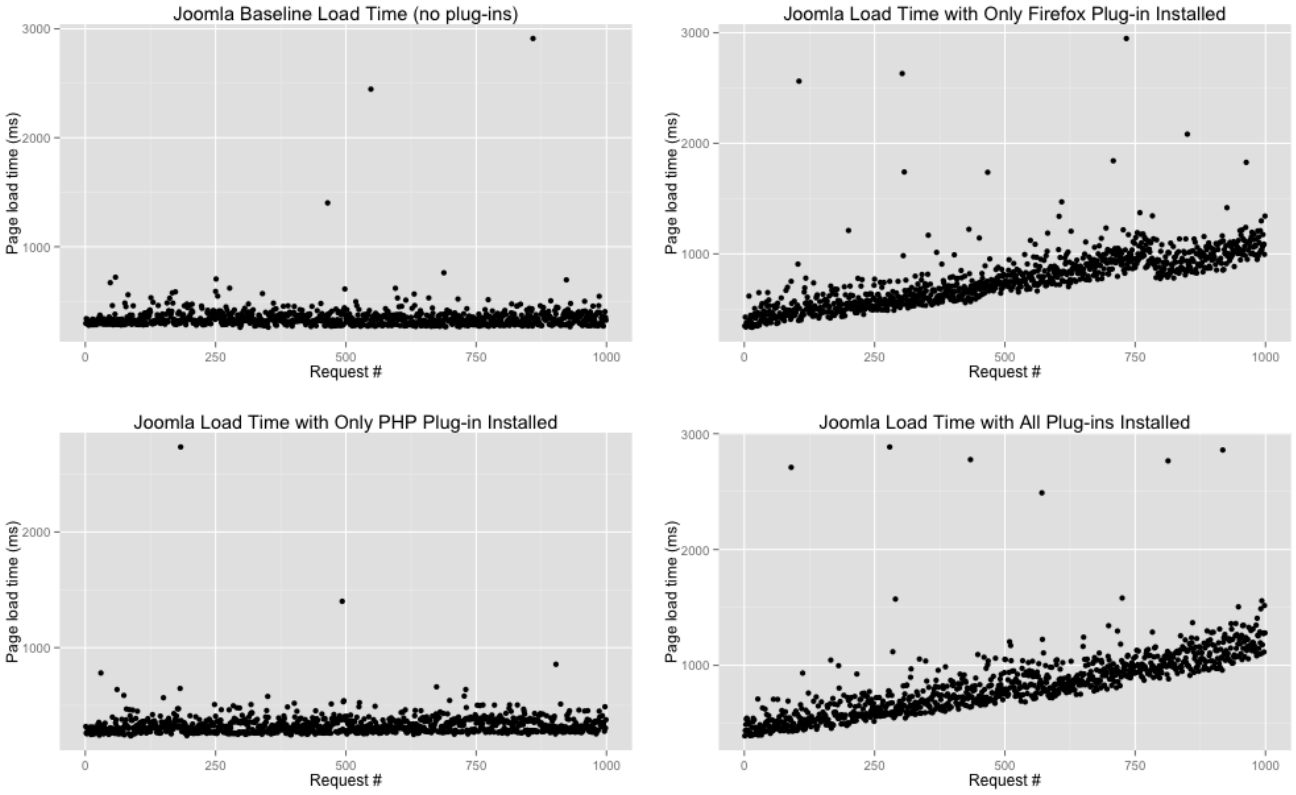


Figure 4.13: Initial performance of MT-WAVE while attempting to measure overhead.

To determine the source of the performance problem, the MT-WAVE Visualizer is used. Inspecting the first and last "all plug-ins enabled" execution traces in the MT-WAVE Visualizer shows that the backend performance remains unchanged, which matches the conclusion one would draw by inspecting the "PHP plug-in only" plot. This is a clear indication that the source of the problem is on the front-end.

In the browser, a context object is created each MT-WAVE-traced page to keep track of in-flight HTTP requests. When a page had completed loading, all of the monitored requests were removed from the context, but the initial context object was not removed because of the possibility of the page making additional requests (e.g. for JSON objects based on user activity). When a new page was loaded, a new context object was created to track the requests for that page, but the original context object was not deleted. By setting up the context objects based on the page, it was impossible to know whether a new page was being loaded in the existing tab, or whether a new tab had been

opened and both pages should be tracked simultaneously. This resulted in a memory leak; context objects were never destroyed. For small experiments this introduced a small bit of latency that went unnoticed until larger experiments were run.

The system was changed to track in-flight requests on a per-tab basis instead of on a per-page basis; since a given tab can only host one page at a time, the solution is to simply clear any existing context objects for a tab when a new page load is requested. The memory leak went away, and tracing performance remained constant even when a large number of samples were recorded.

4.5 Summary of MT-WAVE Performance on Real Applications

In all three performance-enhancement case studies, user-perceived application performance was improved. MT-WAVE assisted this process by providing a clear execution trace of the web applications, and the visualizations made it simple to quickly identify target functions for optimization.

In the Magento and Wordpress cases, the application structure and source code had not been inspected or understood ahead of time; MT-WAVE provided all of the insight required to increase application performance. Additionally, MT-WAVE proved to be a useful tool for understanding application execution. By tracing an application and using the visualizer to quickly step through the execution trace, it was simple to understand which modules and files were used to go from incoming request, to HTML, to fully-formed HTTP response.

In the Social Spiral case, two previously unknown sources of latency were identified. Despite the fact that this is an application that the author helped develop, the two identified defects were in unexpected places. This reinforces the value of Millsap's [40] methodology for performance optimization: "Work first to reduce the biggest response time *component* of a business' most important user action." Without having tools that can accurately measure the response time components of an application, it is impossible to work to reduce their execution time.

CHAPTER 5

CONCLUSIONS & FUTURE WORK

5.1 Summary

This thesis demonstrates both the feasibility and utility of multi-tier tracing for web applications. MT-WAVE is a combination of techniques, tools, and methodology that allows developers to pinpoint the source of performance problems in web applications. This is accomplished by doing low-overhead execution tracing and visualizing the results.

Using the methodology described in Section 4.1.2, a mixture of open source and commercial applications were evaluated using MT-WAVE. This methodology combines Breadcrumb tracing for connecting performance traces between tiers, X-Trace for event aggregation, an in-browser plug-in for instrumenting HTTP requests, and several different techniques for tracing the application code at each tier. To demonstrate the utility of this methodology, it was applied to several open-source and commercial applications.

In the Magento case (Section 4.3.1), several attempts were required to identify a change that would positively affect system performance. As is often the case, code changes for optimization may not always have the intended result; this case study clearly illustrates how this can happen. The final result was a latency reduction of 20%.

For Wordpress (Section 4.3.2), the largest source of latency was caused by reloading and parsing the code from disk for each page load. This problem was addressed by adding a caching module to the web server to preserve the parsed code. This simple configuration change reduced latency by 12%

Finally, the Social Spiral case study (Section 4.3.3) identified two performance problems: latency due to lock contention in the Ruby interpreter and latency due to complicated request processing. While MT-WAVE rapidly identified the lock contention, the solution to that problem is complicated and has the potential to de-stabilize the application's servers; as such, further investigation is left as future work. MT-WAVE effectively identified the complicated request processing and a caching layer was introduced to prevent re-computation of the results for every request. This caching resulted in a latency reduction of 33%.

In all cases, the MT-WAVE visualization system directly guided the optimization process. A baseline experiment is run first, and then high-latency components of the application are identified and targeted for optimization. After a proposed optimization has been implemented, new data is collected and the MT-WAVE traces are compared to verify that a positive change has taken place.

5.2 Conclusions

There are two main contributions in this work:

- A tracing methodology and implementation suitable for doing multi-tier tracing in web applications;
- A visualization system for interpreting execution traces in web applications.

A broad survey of existing methodologies and techniques resulted in a small set of techniques that, when combined, would be applicable to multi-tier tracing for web applications. Described in Section 3.1, a combination of Breadcrumb tracing, X-Trace for event aggregation, in-browser capture for HTTP instrumentation, and a number of techniques for application instrumentation are used.

The visualization system, described in Section 3.4, was used throughout the case studies to analyze the collected execution traces and provide insight and guidance for the optimization process. This system was used to rapidly understand both the structure of the traced application and the source of performance problems; optimization was rapidly performed on unfamiliar source code by simply following the identified slow parts of the execution traces.

Combined, the tracing techniques and visualization system was tested on a number of web applications to determine their utility. By using the methodology and techniques on real web applications, it is shown that they provide an effective set of tools for identifying the cause of performance problems in web applications.

5.3 Future Work

5.3.1 Investigation of Ruby on Rails Threading

The first part of the Social Spiral experiments (Section 4.3.3) identified a challenging performance problem: the application threads were initially waiting on a limited pool of database connections. After growing the connection pool, though, the system performance remained unchanged. This appears to be the result of a Global VM Lock inside the Ruby interpreter. Ruby on Rails is a popular application framework, Providing guidance by doing a thorough investigation of the latency and resource trade-offs between a thread-based and multiprocess-based Rails application servers would benefit both the Rails developer community. Further, many interpreted programming languages have similar issues with multi-threaded code, and this work would likely have broader applicability than just Ruby on Rails.

5.3.2 Improved Aggregation and Algorithms

Currently the MT-WAVE visualization system operates on the execution traces one-at-a-time. All of the aggregation happens using a collection of Ruby scripts to extract key parameters from the execution traces, and R scripts to generate the plots showing the performance over multiple page loads.

There are a number of algorithms (see Section 2.7.6) that are useful for automatically identifying similarities and differences in execution traces. Integrating one of these algorithms into the visualization system could improve the

developer's ability to pinpoint changes in system performance over time. Combining these algorithms with a framework for doing periodic tracing could give developers automatic notification when system performance has changed due to application or configuration changes.

5.3.3 Deeper Integration

This work has focused on proving that it is both feasible and beneficial to do multi-tier tracing in web applications, by building a set of example tracing plug-ins for common open-source programming languages. This set of plug-ins does not yet extend to the full depths of a web application stack. For example, it would be beneficial to integrate with common databases to provide visibility into the internal processing, such as automatically extracting query plans and result set sizes.

5.3.4 Crossing Administrative Domains

In the standard implementation of X-Trace, there is poor support for crossing administrative domains; traces are aggregated to a single X-Trace server controlled by a single organization. A common feature of web applications is to use APIs provided by varying providers. For example, Social Spiral allows users to authenticate using their Facebook or Twitter accounts. As applications continue to integrated with third-party services, performance problems will become more challenging to debug; instead of just trying to identify which request and functions are responsible for the unacceptable latency, developers will first have to determine which service is responsible.

BIBLIOGRAPHY

[1] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *19th ACM Symposium on Operating Systems Principles*, pages 74–89, Bolton Landing, NY, October 2003.

[2] M.J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, 1986.

[3] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for Request Extraction and Workload Modelling. In *6th Symposium on Operating Systems Design & Implementation*, pages 259–272, San Francisco, CA, December 2004.

[4] Adam Barth. HTTP State Management Mechanism. RFC 6265, April 2011.

[5] Bert Bos, Tantek Celik, Ian Hickson, and Hakon Wium Lie. Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification, Appendix G: Grammar of CSS 2.1. W3C Recommendation, June 2011.

[6] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eve Maler, and Francois Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C Recommendation, November 2008.

[7] Ethan Brown. *Web Development with Node & Express*. O'Reilly Media, Sebastopol, CA, July 2014.

[8] Anupam Chanda, Alan L. Cox, and Willy Zwaenepoel. Whodunit: Transactional Profiling for Multi-Tier Applications. In *2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys'07)*, pages 17–30, Lisbon, Portugal, March 2007.

[9] Andre Charland and Brian Leroux. Mobile application development: web vs. native. *Communications of the ACM*, 54:49–53, April 2011.

[10] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *2002 International Conference on Dependable Systems and Networks*, pages 595–604, Bethesda, MD, June 2002.

[11] Tom Christiansen, Brian Foy, Larry Wall, and Jon Orwant. *Programming Perl, 4th Edition*. O'Reilly & Associates, Inc., Sebastopol, CA, February 2012.

[12] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[13] Jesse Cravens and Thomas Q Brady. *Building Web Apps with Ember.js*. O'Reilly Media, Sebastopol, CA, July 2014.

[14] Samuel Dauzon. *Django Essentials*. Packt Publishing, Birmingham, UK, June 2014.

[15] Caleb Doxsey. *An introduction to programming in Go*. CreateSpace Independent Publishing, Scotts Valley, CA, September 2012.

[16] Ulfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. Fay: Extensible distributed tracing from kernels to clusters. *ACM Transactions on Computer Systems (TOCS)*, 30(4), 2012.

[17] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10(1):56–66, 1988.

[18] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: A Pervasive Network Tracing Framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation (NSDI'07)*, pages 271–284, Cambridge, MA, April 2007.

[19] Susan L. Graham, Peter B Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. In *1982 SIGPLAN symposium on Compiler construction*, pages 120–126, Boston, MA, 1982.

[20] Brendan Gregg. Thinking methodically about performance. *Communications of the ACM*, 56(2):45–51, February 2013.

[21] Adam Griffiths. *CodeIgniter 1.7 Professional Development*. Packt Publishing, Birmingham, UK, April 2010.

[22] Miguel Grinberg. *Flask Web Development: Developing Web Applications with Python*. O'Reilly Media, Sebastopol, CA, April 2014.

[23] Joseph L. Hellerstein, Mark M. Maccabee, W. Nathaniel Mills III, and John J. Turek. ETE: A Customizable Approach to Measuring End-to-End Response Times and Their Components in Distributed Systems. In *19th IEEE International Conference on Distributed Computing Systems (ICDCS'99)*, pages 152–162, Austin, TX, May 1999.

[24] Ian Hickson, Robin Berjon, Steve Faulkner, Travis Leithead, Erika Navara, Edward O'Connor, and Silvia Pfeiffer. HTML5: A Vocabulary and associated APIs for HTML and XHTML. W3C Recommendation, October 2014.

[25] Ecma International. Standard ECMA-262: ECMAScript Language Specification, 5.1 Edition. `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`, June 2011.

[26] Ecma International. Standard ECMA-404: The JSON Data Interchange Format. `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`, October 2013.

[27] Deepal Jayasinghe, Simon Malkowski, Qingyang Wang, Jack Li, Pengcheng Xiong, and Calton Pu. Variations in performance and scalability when migrating n-tier applications to different clouds. In *2011 IEEE Conference on Cloud Computing (CLOUD 2011)*, pages 73–80, Washington, DC, July 2011.

[28] Emre Kiciman and Benjamin Livshits. Ajaxscope: A platform for remotely monitoring the client-side behavior of web 2.0 applications. *ACM Trans. Web*, 4(4):13:1–13:52, September 2010.

[29] Eric Koskinen and John Jannotti. BorderPatrol: isolating events for black-box tracing. In *3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 191–203, Glasgow, Scotland, March 2008.

[30] Darja Krushevskaja and Mark Sandler. Understanding latency variations of black box services. In *Proceedings of the 22nd international conference on World Wide Web*, pages 703–714, Rio de Janeiro, Brazil, May 2013.

[31] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, July 1978.

[32] Myungjin Lee, Ramana Rao Kompella, and Sumeet Singh. Ajaxtracker: Active measurement system for high-fidelity characterization of ajax applications. In *2010 USENIX Conference on Web Application Development (WebApps'10)*, Boston, MA, June 2010.

[33] Zhichun Li, Ming Zhang, Zhaosheng Zhu, Yan Chen, Albert Greenberg, and Yi-Min Wang. Webprophet: Automating performance prediction for web services. In *7th USENIX Conference on Networked Systems Design and Implementation (NSDI'10*, pages 10–10, San Jose, California, April 2010.

[34] Bill Lubanovic. *Introducing Python*. O'Reilly & Associates, Inc., Sebastopol, CA, November 2014.

[35] Gideon Mann, Mark Sandler, Darja Krushevskaja, Sudipto Guha, and Eyal Even-Dar. Modeling the parallel execution of black-box services. In *3rd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'11)*, pages 20–20, Portland, OR, June 2011.

[36] Keith Ansel Marzullo. *Maintaining the time in a distributed system: an example of a loosely-coupled distributed service (synchronization, fault-tolerance, debugging)*. PhD thesis, 1984.

[37] Friedemann Mattern. Virtual time and global states of distributed systems. In *Workshop on Parallel and Distributed Algorithms*, pages 215–226, Chateau De Bonas, France, May 1989.

[38] Patrick Meenan. How fast is your website? *Commun. ACM*, 56(4):49–55, April 2013.

[39] Remigius Meier and Armin Rigo. A Way Forward in Parallelising Dynamic Languages. In *9th Implementation, Compilation, Optimization of OO Languages, Programs and Systems Workshop (ICOOOLPS'14)*, pages 4:1–4:4, Uppsala, Sweden, July 2014.

[40] Cary Millsap and Jeff Holt. *Optimizing Oracle Performance*. O'Reilly & Associates, Inc., Sebastopol, CA, 2003.

[41] Clement Nedelcu. *Nginx HTTP Server, 2nd Edition*. O'Reilly Media, Sebastopol, CA, July 2013.

[42] Jakob Nielsen. Powers of 10: Time Scales in User Experience. `http://www.useit.com/alertbox/timeframes.html`, October 2009.

[43] Addy Osmani. *Developing Backbone.js Applications*. O'Reilly Media, Sebastopol, CA, May 2013.

[44] Mark Purdy. Modern performance monitoring. *ACM QUEUE*, 4, February 2006.

[45] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. Appinsight: Mobile app performance monitoring in the wild. In *10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*, pages 107–120, Hollywood, CA, USA, October 2012.

[46] Rick Reed. That's 'Billion' With a 'B': Scaling to the Next Level at WhatsApp. Erlang Factory 2014, March 2014.

[47] Leon Revill. *jQuery 2.0 Development Cookbook*. Packt Publishing, Birmingham, UK, February 2014.

[48] Sam Ruby, Dave Thomas, and David Heinemeier Hansson. *Agile Web Development with Rails 3.2, 4th Edition*. Pragmatic Bookshelf, Raleigh, NC, April 2011.

[49] Raja Sambasivan, Alice Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory Ganger. Diagnosing performance changes by comparing request flows. In *8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*, March 2011.

[50] Eric Schrock. Debugging ajax in production. *Commun. ACM*, 52(5):57–60, May 2009.

[51] Shyam Seshadri and Brad Green. *AngularJS: Up and Running*. O'Reilly Media, Sebastopol, CA, September 2014.

[52] Ben Shneiderman. The eyes have it: a task by data type taxonomy for information visualizations. In *IEEE Symposium on Visual Languages*, pages 336–343, Boulder, CO, September 1996.

[53] Eric Shurman and Jake Brutlag. Performance Related Changes and their User Impact. Conference presentation, June 2009.

[54] Benjamin H. Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donal Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. `http://research.google.com/pubs/pub36356.html`, 2010.

[55] Kevin Tatroe, Peter MacIntyre, and Rasmus Lerdorf. *Programming PHP, 3rd Edition*. O'Reilly & Associates, Inc., Sebastopol, CA, February 2013.

[56] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R. Ganger. Stardust: Tracking Activity in a Distributed Storage System. In *Joint International Conference*

on Measurement and Modeling of Computer Systems (SIGMETRICS'06), pages 3–14, Saint Malo, France, June 2006.

[57] Dave Thomas, Andy Hunt, and Chad Fowler. *Programming Ruby 1.9 & 2.0*. Pragmatic Bookshelf, Raleigh, NC, June 2013.

[58] Andrej van der Zee, Alexandre Courbot, and Tatsuo Nakajima. mBrace: action-based performance monitoring of multi-tier web applications. In *3rd Workshop on Dependable Distributed Data Management (WDDM'09)*, pages 29–32, Nuremberg, Germany, March 2009.

[59] Anne van Kesteren, Julian Aubourg, Jungkee Song, and Hallvord R. M. Steen. XMLHttpRequest. W3C Recommendation, December 2012.

[60] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. Demystifying page load performance with wprof. In *10th USENIX Conference on Networked Systems Design and Implementation (NSDI'13)*, pages 473–486, Lombard, IL, April 2013.

[61] Web Hypertext Application Technology Working Group (WHATWG). HTML - Living Standard. `http://www.whatwg.org/specs/web-apps/current-work/multipage/parsing.html`, June 2013.

[62] Joe Yates. *Instant Sinatra Starter*. Packt Publishing, Birmingham, UK, June 2013.

CHAPTER A
SOURCE CODE

A.1 PHP Tracing

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif
#include <stdio.h>
#include <time.h>
#include <sys/time.h>
#include "php.h"
#include "main/php_version.h"
#include "zend_extensions.h"
#include "Zend/zend_API.h"
#include "php_mtwave.h"
#include "apd.h"

#include "xtrmd2.h"

#define BUF_SIZE 2048

static function_entry mtwave_functions[] = {
  PHP_FE(hello_mtwave, NULL)
  {NULL, NULL, NULL}
};

zend_module_entry mtwave_module_entry = {
  STANDARD_MODULE_HEADER,
  PHP_MTWAVE_EXTNAME,
  mtwave_functions,
  PHP_MINIT(mtwave),
  PHP_MSHUTDOWN(mtwave),
  PHP_RINIT(mtwave),
  PHP_RSHUTDOWN(mtwave),
  PHP_MINFO(mtwave),
  PHP_MTWAVE_VERSION,
  NO_MODULE_GLOBALS,
  NULL,
  STANDARD_MODULE_PROPERTIES_EX
};


#ifdef COMPILE_DL_MTWAVE
```

```
ZEND_GET_MODULE(mtwave)
#endif

ZEND_DECLARE_MODULE_GLOBALS(mtwave);

static char log_prefix[256];

static void mtlog(char* msg) {
  char buf[BUF_SIZE];
  int res;
  struct sockaddr_un sun;

  struct timeval tv;
  if (XG(log_fileno) == -1) {
    XG(log_fileno) = socket(AF_UNIX, SOCK_STREAM, 0);
    sun.sun_family = AF_UNIX;
    strcpy(sun.sun_path, "/tmp/xtrace_proxy");
    res = connect( XG(log_fileno), (struct sockaddr*)&sun,
      sizeof(struct sockaddr_un));
    if (res == -1) {
      perror("connect");
      close( XG(log_fileno));
      XG(log_fileno) = -1;
    }
  }

  if(XG(log_fileno) != -1) {
    gettimeofday(&tv, NULL);
    sprintf( buf, "[%u.%06u]%s%s", tv.tv_sec, tv.tv_usec,
      log_prefix, msg);
    assert(strlen(buf) < BUF_SIZE);
    res = send( XG(log_fileno), buf, strlen(buf), 0 );
    if (res == -1) {
      perror("send");
      close( XG(log_fileno));
      XG(log_fileno) = -1;
    }
  }
}

char* xtrace_message(char* xtrmd, char* agent, char* edge,
    char* label, struct timeval timestamp) {
  char* buf;
  char* new_buf;
  int current_size = 256;
  int written_size = 0;

  buf = (char*)emalloc(current_size);

  do {
    written_size = snprintf(buf, current_size,
                "X-Trace Report ver 1.0\n"
                "X-Trace: %s\n"
                "Agent: %s\n"
```

```
                "Label: %s\n"
                "Timestamp: %d.%06d\n"
                "Edge: %s\n",
                xtrmd, agent, label, timestamp.tv_sec,
                timestamp.tv_usec,
                edge);
    if (written_size >= (current_size - 1)) {
      new_buf = erealloc(buf, written_size+1);
      if (new_buf) {
    buf = new_buf;
      } else {
    efree(buf);
      }
    } else {
      break;
    }
  } while(1);
  return buf;
}

static void increase_log_indent() {
  strcat(log_prefix, "  ");
  //  xtrace_push_context( &XG(xtrace) );
}

static void decrease_log_indent() {
  int new_len = strlen(log_prefix) - 2;
  new_len = (new_len < 0) ? 0 : new_len;
  log_prefix[new_len] = 0;
  // xtrace_pop_context( &XG(xtrace) );
}

PHP_FUNCTION(hello_mtwave)
{
  int t = time(NULL);
  char* s = emalloc(255);
  sprintf(s, "Hello World (%d) (%d)\n", XG(init_status), t);
  RETURN_STRING(s, 0);
}


void (*old_execute)(zend_op_array *op_array TSRMLS_DC);
void (*old_execute_internal)(zend_execute_data *execute_data_ptr,
  int return_value_used TSRMLS_DC);
void mtwave_execute(zend_op_array *op_array TSRMLS_DC);
void mtwave_execute_internal(zend_execute_data *op_array,
  int return_value_used TSRMLS_DC);

PHP_MINIT_FUNCTION(mtwave) {
  old_execute = zend_execute;
  old_execute_internal = zend_execute_internal;

  zend_execute = mtwave_execute;
  zend_execute_internal = mtwave_execute_internal;
```

```
  log_prefix[0] = 0;
  XG(init_status) = 0;
  XG(log_fileno) = -1;
  XG(mtwave_active) = 0;
  XG(request_count) = 0;

  return SUCCESS;
}

PHP_MSHUTDOWN_FUNCTION(mtwave) {
  zend_execute = old_execute;
  zend_execute_internal = old_execute_internal;
  return SUCCESS;
}

void mtwave_execute(zend_op_array *op_array TSRMLS_DC) {
  if( ! XG(mtwave_active) ) {
    old_execute (op_array TSRMLS_CC);
    return;
  }

  int lineno;
  char* file;
  char* function;

  zend_function* func = (zend_function*) op_array;

  TSRMLS_FETCH();  /* load thread-safe resource manager */

  file = zend_get_executed_filename(TSRMLS_C);
  lineno = zend_get_executed_lineno(TSRMLS_C);
  function = apd_get_active_function_name(op_array TSRMLS_C);

  if (!function) {
    function = "{main}";
  }

  char buf[BUF_SIZE];

  sprintf(buf, "exec [%s] (%s:%d)\n", function,
    file, lineno);

  efree(function);

  assert(strlen(buf) < BUF_SIZE);
  mtlog(buf);
  increase_log_indent();
  old_execute (op_array TSRMLS_CC);
  decrease_log_indent();
  mtlog("end_exec\n");
}

void mtwave_execute_internal(zend_execute_data *execute_data,
    int return_value_used TSRMLS_DC) {
```

```
  char buf[BUF_SIZE];
  char* name;

  if( ! XG(mtwave_active) ) {
    if (old_execute_internal) {
      old_execute_internal(execute_data, return_value_used);
    } else {
      execute_internal(execute_data, return_value_used);
    }
    return;
  }

  name = ((zend_internal_function*)execute_data
          ->function_state.function)->function_name;
  //  xtrace_push_context( &XG(xtrace) );
  // xtrace_set_function( &XG(xtrace), name, "<internal>", 0);

  sprintf(buf, "exec_int [%s]\n", name);
  assert(strlen(buf) < BUF_SIZE);
  mtlog(buf);
  if (old_execute_internal) {
    old_execute_internal(execute_data, return_value_used);
  } else {
    execute_internal(execute_data, return_value_used);
  }
  mtlog("end_exec_int\n");
  // xtrace_pop_context( &XG(xtrace) );
}

PHP_MINFO_FUNCTION(mtwave) {
  php_info_print_table_start();
  php_info_print_table_header(2, "mtwave support", "enabled");
  php_info_print_table_row(2, "Version", PHP_MTWAVE_VERSION);
  php_info_print_table_end();
}

/*********************************************************
 * Request handler part
 */


PHP_RINIT_FUNCTION(mtwave) {
  /* look for mtwave header */
  zval** mtwave_header;
  char buf[BUF_SIZE];
  XTRMD xtrmd_result;
  int res;
  char branch_id[20];
  char trace_plan_id[20];

  XG(request_count)++;
  XG(log_fileno) = -1;

  XG(init_status) = 1;
```

```
  zend_is_auto_global("_SERVER", sizeof("_SERVER")-1 TSRMLS_CC);

  if ( zend_hash_find(
         Z_ARRVAL_P(PG(http_globals)[TRACK_VARS_SERVER]),
         "HTTP_X_TRACEFF", 15,
         (void**)&mtwave_header) != FAILURE) {
    sprintf(buf, "RINIT: found header: %s\n",
        Z_STRVAL_PP(mtwave_header));
    assert(strlen(buf) < BUF_SIZE);
    mtlog(buf);
    XG(mtwave_active) = 1;

    res = xtrmd_parse( Z_STRVAL_PP(mtwave_header), &xtrmd_result );
    if (res == 0) {
      sprintf(buf, "RINIT: task %s op %s options %s\n",
          xtrmd_result.taskId, xtrmd_result.opId,
          ( xtrmd_result.flags.options ?
            xtrmd_result.options : "[none]"));
      assert(strlen(buf) < BUF_SIZE);
      mtlog(buf);

      res = xtrmd_find_option(&xtrmd_result, "80", branch_id);
      if ( res == 0 ) {
    sprintf(buf, "RINIT: branch id %s\n", branch_id);
    assert(strlen(buf) < BUF_SIZE);
    mtlog(buf);
      }

    } else {
      mtlog("XTRMD: Parse error\n");
    }

    xtrace_init( &XG(xtrace) );
  } else {
    XG(mtwave_active) = 0;
  }

  return SUCCESS;
}

PHP_RSHUTDOWN_FUNCTION(mtwave) {

  if ( XG(mtwave_active) ) {
    mtlog("PHP_RSHUTDOWN_FUNCTION\n");
    xtrace_teardown( &XG(xtrace) );
  }
  if ( XG(log_fileno) ) {
    close( XG(log_fileno) );
    XG(log_fileno) = -1;
  }
}

/*************************************************************
```

```
 * Tracing code
 */

ZEND_DLEXPORT void mtwave_statement_call(zend_op_array* op_array) {
  int lineno;
  char* file;
  zend_function* func = (zend_function*) op_array;

  if( !XG(mtwave_active) ) {
    return;
  }

  TSRMLS_FETCH();  /* load thread-safe resource manager */

  lineno = EG(current_execute_data)->opline->lineno;
  file = op_array->filename;

  char buf[BUF_SIZE];
  sprintf(buf,
    "stmt: %s (%s:%d)\n", get_active_function_name(TSRMLS_C),
    file, lineno);
  assert(strlen(buf) < BUF_SIZE);
  mtlog(buf);
}


/*************************************************************
 * Zend Extension Part
 */

ZEND_DLEXPORT int mtwave_zend_startup(zend_extension* ext) {
  //CG(extended_info) = 1; /* turn on the profiling code */
  return zend_startup_module(&mtwave_module_entry);
}

ZEND_DLEXPORT void mtwave_zend_shutdown(zend_extension* ext) {
  /* Do nothing. */
}

#ifndef ZEND_EXT_API
#define ZEND_EXT_API    ZEND_DLEXPORT
#endif
ZEND_EXTENSION();

ZEND_DLEXPORT zend_extension zend_extension_entry = {
  PHP_MTWAVE_EXTNAME,
  PHP_MTWAVE_VERSION,
  "Anthony Arkles",
  "none",
  "Copyright 2010",
  mtwave_zend_startup,
  mtwave_zend_shutdown,
  NULL,           /* activate_func_t */
  NULL,           /* deactivate_func_t */
```

```
  NULL,           /* message_handler_func_t */
  NULL,           /* op_array_handler_func_t */
  NULL,           /* was statement_handler_func_t */
  //  mtwave_statement_call, /* statement_handler_func_t */
  NULL, /* fcall_begin_handler_func_t - was begin func*/
  NULL,  /* fcall_end_handler_func_t - was end func*/
  NULL,   /* op_array_ctor_func_t */
  NULL,           /* op_array_dtor_func_t */
  STANDARD_ZEND_EXTENSION_PROPERTIES
};
```

A.2 Firefox Tracing Plug-in

```
Components.utils.import("resource://gre/modules/XPCOMUtils.jsm");

const CLASS_ID =
  Components.ID("{4f9c1960-1f27-11df-8a39-0800200c9a66}");
const CONTRACT_ID = "@chilly.ca/xtraceff-service;1";
const CLASS_NAME = "XTraceFF Javascript XPCOM Component";
const nsISupports = Components.interfaces.nsISupports;
const nsIHttpChannel = Components.interfaces.nsIHttpChannel;
const LOAD_DOCUMENT_URI =
  Components.interfaces.nsIChannel.LOAD_DOCUMENT_URI;

function log_debug(msg) {
  dump(msg);
  return;
}

function XTraceFF_Service() {
  this.wrappedJSObject = this;
  this.contexts = [];
  this.is_bound = false;

}

XTraceFF_Service.prototype = {
  classID: CLASS_ID,
  bind : function() {
    if(this.is_bound) {
      return;
    }
    this.is_bound = true;
    this.hro = new XTraceFF_HttpRequestObserver(this);
    this.hro.bind();
  },
  hello : function() {
    log_debug("Hello World\n");
  },
  getContextIndex : function(target) {
    for(var i in this.contexts) {
      if(this.contexts[i]['target'] == target) {
        return i;
```

```
    }
  }
  return -1;
},
findContext : function(target) {
  var idx = this.getContextIndex(target);
  return idx >= 0 ? this.contexts[idx] : undefined;
},
deleteContext : function(target) {
  var idx = this.getContextIndex(target);
  if (idx != -1) {
    this.contexts.splice(idx, 1);
  }
},
get_default_mtwave_enabled_state : function() {
  var prefs = Components.classes[
    "@mozilla.org/preferences-service;1"]
    .getService(Components.interfaces.nsIPrefService);
  prefs = prefs.getBranch("extensions.mtwave1.");

  return prefs.getBoolPref('on_at_startup');
},

createContext : function(target) {
  this.contexts.push(
    {
          enabled : this.get_default_mtwave_enabled_state(),
          target : target,
          xtrmd : new XTraceFF_XtrContext(),
          owner_channel : undefined,
      requests: []
    });
},
destroyContext : function(target) {
  this.deleteContext(target);
},
isXtraceEnabled : function(target) {
  var context = this.findContext(target);
  if(!context) {
    return undefined;
  }
  if(context.enabled) {
    return true;
  } else {
    return false;
  }
},
setXtraceEnabled : function(target, value) {
  var context = this.findContext(target);
  context.enabled = value;
},

////////////////////////////////////////////////////////
// code for interacting with HTTP channels (nsIHttpChannel)
```

```
//
// getBrowserFromChannel is due to
// https://developer.mozilla.org/En/Code_snippets/Tabbed_browser
getRequestWebProgress : function(request) {
  // copied from Firebug.  The Mozilla docs suck.
  request.QueryInterface(Components.interfaces.nsIHttpChannel);
  try {
    if (request && request.notificationCallbacks) {
      var v =request.notificationCallbacks.QueryInterface(
        Components.interfaces.nsIInterfaceRequestor).getInterface(
        Components.interfaces.nsIWebProgress);
      return v;
    }
  } catch (x) {

  }

  try {
    if(request && request.loadGroup &&
        request.loadGroup.groupObserver) {
      var v = request.loadGroup.groupObserver.QueryInterface(
        Components.interfaces.nsIInterfaceRequestor).getInterface(
        Components.interfaces.nsIWebProgress);
      return v;
    }
  } catch (x) {
  }
  log_debug("==== No success, returning null\n");
  return null;
},
getWindowForRequest: function(request) {
  var context = this.getContextForRequest(request);
  if (context) {
    try {
      return context.associatedWindow;
    } catch(exc) {
      // this will occasionally bomb out when
      // accessing things we don't care about
    }
  }
  return null;
},
getContextForRequest: function(request) {
  var context;
  try {
    return (request.notificationCallbacks
      .getInterface(Components.interfaces.nsILoadContext));
  } catch(exc) {
  }

  try {
    return (request.loadGroup.notificationCallbacks
      .getInterface(Components.interfaces.nsILoadContext);
  } catch(exc) {
```

```
    }

  },
  getBrowserFromChannel: function (aChannel) {
    try {
      var domWin = this.getWindowForRequest(aChannel);
      var wm = Components.classes[
        "@mozilla.org/appshell/window-mediator;1"]
        .getService(Components.interfaces.nsIWindowMediator);
      var enumerator = wm.getEnumerator(null);
      var browser = null;
      while(enumerator.hasMoreElements()){
            var current_browser = enumerator.getNext();
            var browser_attempt;
            try {
              browser_attempt = current_browser.gBrowser
            .getBrowserForDocument(domWin.top.document);
            } catch (x) {
              // nothing
            }
            if (browser_attempt) {
              browser = browser_attempt;
              break;
            }
      }
      return browser;
    }
    catch (e) {
      log_debug("%%% " +e + "\n");
      log_debug("^^^^^^ " + aChannel.URI.spec + "\n");
      return null;
    }
  },

  QueryInterface : function(aIID) {
    if (!aIID.equals(nsISupports))
      throw Components.results.NS_ERROR_NO_INTERFACE;
    return this;
  }
};

const NSGetFactory = XPCOMUtils.generateNSGetFactory(
  [ XTraceFF_Service ]);

////////////////////////////////////////////////////////////
// XTraceFF Service - HTTP Observer
//
// This is porting the XTraceFF.Engine.HttpRequestObserver
// code over
// into the XPCOM component -- no more Ci and Cc problems
// from having this missing
//
////////////////////////////////////////////////////////////
```

```
XTraceFF_HttpRequestObserver = function(XTRS) {
  this.XTRS = XTRS;
  this.init();
}

XTraceFF_HttpRequestObserver.prototype = {
  init : function() {
    this.current_connections = [];
    this.timer = Components.classes["@mozilla.org/timer;1"]
      .createInstance(Components.interfaces.nsITimer);
    var that = this;
    var cbk = {
      notify: function() {
        that.length;
      }
    };
    this.timer.initWithCallback(cbk, 2000,
      Components.interfaces.nsITimer.TYPE_REPEATING_SLACK);
  },
  bind : function() {
    var Cc = Components.classes;
    var Ci = Components.interfaces;

    var observerService = Cc["@mozilla.org/observer-service;1"]
      .getService(Ci.nsIObserverService);
    observerService.addObserver(this, "http-on-modify-request",
      false);
    observerService.addObserver(this, "http-on-examine-response",
      false);
    observerService.addObserver(this,
      "http-on-examine-cached-response", false);
    observerService.addObserver(this,
      "http-on-examine-merged-response", false);
    var distributorService =
      Cc["@mozilla.org/network/http-activity-distributor;1"]
      .getService(Ci.nsIHttpActivityDistributor);
    distributorService.addObserver(this);

  },
  unbind : function() {
  },
  browser_for_subject : function( subject ) {
    subject.QueryInterface(Components.interfaces.nsIChannel);
    var browser = this.XTRS.getBrowserFromChannel(subject);
    return browser;
  },
  should_observe : function( subject ) {
    var prefs = Components.classes[
      "@mozilla.org/preferences-service;1"]
      .getService(Components.interfaces.nsIPrefService);
    prefs = prefs.getBranch("extensions.mtwave1.");

    try {
      subject.QueryInterface(nsIHttpChannel);
```

```
    /* don't capture our own outbound XMLHttpRequets */
    if(subject.URI.asciiSpec.indexOf(
        prefs.getCharPref('mtwave_server')) == 0) {
          return false;
    }

    var target = "http://localhost:4444/";
    if(subject.URI.asciiSpec.substr(0, target.length) == target) {
          return false;
    }
    /* this might capture webdriver traffic? */
    target = "http://127.0.0.1:4444/";
    if(subject.URI.asciiSpec.substr(0, target.length) == target) {
          return false;
    }
  } catch (x) {
    log_debug(x);
  }

  var browser = this.browser_for_subject(subject);
  if(browser != null && this.XTRS.isXtraceEnabled(browser)) {
    return true;
  }
  return false;
},
context_for_subject : function( subject ) {
  var browser = this.browser_for_subject(subject);
  if(!browser) { return null; }
  return this.XTRS.findContext(browser);
},

// Implementation of nsIObserver
observe : function( subject, topic, data) {

  if (!this.should_observe(subject)) {
    return;
  }

  this.handle_monitored_request(subject, topic, data);
},

// Implementation of nsIHttpActivityObserver
observeActivity : function(aHttpChannel, aActivityType,
    aActivitySubtype, aTimestamp, aExtraSizeData,
    aExtraStringData) {
  if(!this.should_observe(aHttpChannel)) {
    return;
  }

  var request = this.get_request(aHttpChannel, false);
  if (request) {
    var activityStr = this.get_activity_string_for_activity(
      aActivityType, aActivitySubtype);
    if(!activityStr) {
```

```
      return;
    }
    // aTimestamp is in microseconds...
    request.model
      .log_activity( activityStr, aTimestamp / 1000000.0);
  }
},

handle_monitored_request : function( subject, topic, data ) {
  if(topic === 'http-on-modify-request') {
    this.handle_modify_request(subject);
  }
  else if((topic === "http-on-examine-response") ||
          (topic === "http-on-examine-cached-response") ||
          (topic === "http-on-examine-merged-response")) {
    var popped = this.get_request(subject, false);
    if(popped) {
      var win = this.XTRS.getWindowForRequest(subject);
          var context = this.context_for_subject(subject);
      win.wrappedJSObject.xtrace_taskId = context.xtrmd.taskId;
          var xtrmd = popped.model.xtrmd;
          var options = {     };
          options['Epoch'] = context.epoch;
          if(topic === 'http-on-examine-cached-response') {
            options['Cached'] = "true";
          }
          if(popped.model.branch_id) {
            options['BranchJoin'] = popped.model.branch_id;
          }
          xtrmd.log_event(
            "HttpRequestObserver",
            "End page request for: " + subject.URI.asciiSpec,
            options);
    } else {
          log_debug("Invalid request\n");
    }
  }
  else {

  }
},

get_tags_from_settings : function() {
  var tags, tag, tags_stripped, i;
  var prefs = Components.classes[
    "@mozilla.org/preferences-service;1"]
    .getService(Components.interfaces.nsIPrefService);
  prefs = prefs.getBranch("extensions.mtwave1.");

  try {
    tags = prefs.getCharPref("tags").split(',');
    tags_stripped = [];
    for(i in tags) {
      if (tags.hasOwnProperty(i)) {
```

```
        tag = tags[i];
        tags_stripped.push( tag.replace(/(^\s*)|(\s*$)/g, "");
      }
    }
    log_debug("*** TAGS: " + tags_stripped.join(','));
    return tags_stripped;
  } catch (x) {
    log_debug("*** EXCEPTION: " + x + "\n");
  }
  return null;
},

get_header_transmit_option : function() {
  var prefs = Components.classes[
    "@mozilla.org/preferences-service;1"]
    .getService(Components.interfaces.nsIPrefService);
  prefs = prefs.getBranch("extensions.mtwave1.");

  return prefs.getBoolPref('transmit_headers');
},

handle_modify_request : function( subject, topic, data ) {
  var context = this.context_for_subject(subject);
  subject = subject.QueryInterface(
    Components.interfaces.nsIHttpChannel);

  var window = this.XTRS.getWindowForRequest(subject);

  if((subject.loadFlags & LOAD_DOCUMENT_URI) &&
      subject.loadGroup &&
      subject.loadGroup.groupObserver &&
      window == window.parent) {

    this.clear_all_requests(subject);
    context.xtrmd = new XTraceFF_XtrContext();
    //window.xtrace_task_id = context.xtrmd.taskId;
    context.epoch = (new Date()).getTime() / 1000;
    context.xtrmd.log_event(
      "HttpRequestObserver",
      "Load Document request for: " + subject.originalURI.asciiSpec,
      { 'Epoch' : context.epoch,
        'Tag' : this.get_tags_from_settings(),
        'Title' : subject.originalURI.path
      }, true);
  }

  var xtrmd = context.xtrmd.fork_event_chain();
  xtrmd.log_event(
    "HttpRequestObserver",
    "Page request for: " + subject.originalURI.asciiSpec,
    { 'Epoch' : context.epoch });
  var subj = subject.QueryInterface(
        Components.interfaces.nsIHttpChannel);
```

```
  // rely on the side-effects of this to generate a request object
  var request = this.get_request(subject, true);
  request.model.xtrmd = xtrmd;

  var branch_info = request.model.xtrmd.generate_branch();
  request.model.branch_id = branch_info.branch_id;

  if (this.get_header_transmit_option()) {
    subj.setRequestHeader('X-TraceFF', branch_info.xtrmd_string,
        false);
  }
},

get_activity_string_for_activity : function(activity_type,
    activity_subtype) {
  var hao = Components.interfaces.nsIHttpActivityObserver;
  var st = Components.interfaces.nsISocketTransport;
  switch(activity_type) {
  case hao.ACTIVITY_TYPE_HTTP_TRANSACTION:
    switch(activity_subtype) {
    case hao.ACTIVITY_SUBTYPE_REQUEST_HEADER:
      return "header-queued";
    case hao.ACTIVITY_SUBTYPE_REQUEST_SENT:
      return 'header-sent';
    case hao.ACTIVITY_SUBTYPE_RESPONSE_START:
      return 'response-start';
    case hao.ACTIVITY_SUBTYPE_RESPONSE_HEADER:
      return 'response-header-recvd';
    case hao.ACTIVITY_SUBTYPE_RESPONSE_COMPLETE:
      return 'response-recvd';
    case hao.ACTIVITY_SUBTYPE_TRANSACTION_CLOSE:
      return 'closed';
    }
  case hao.ACTIVITY_TYPE_SOCKET_TRANSPORT:
    switch(activity_subtype) {
    case st.STATUS_RESOLVING: return "resolving";
    case st.STATUS_CONNECTING_TO: return "connecting-to";
    case st.STATUS_CONNECTED_TO: return "connected-to";
    case st.STATUS_SENDING_TO: return "sending-to";
    case st.STATUS_WAITING_FOR: return "waiting-for";
    case st.STATUS_RECEIVING_FROM: return "receiving-from";
    default:
      return "socket-unknown-" + activity_subtype;
    }
  }
  return "Type " + activity_type + " Subtype " + activity_subtype;
  return "";
},
//////////////////////////////////////
// data structures for holding current connection information
//
// get_request returns the data object associated with the
// httpChannel, creating one if necessary
//
```

```
  // drop_request removes the data object from the list
  //
  // find_request returns the index of the httpChannel in the
  // requests array, returning -1 if not found.  (Internal Use Only)
  find_request : function( httpChannel ) {
    var ctx = this.context_for_subject(httpChannel);

    for(var i in ctx.requests) {
      if(ctx.requests[i].http_channel === httpChannel) {
            return i;
      }
    }
    return -1;
  },
  get_request : function( httpChannel, inflate ) {
    var ctx = this.context_for_subject(httpChannel);
    var idx = this.find_request(httpChannel);
    if(idx != -1) {
      return ctx.requests[idx];
    }
    if (inflate) {
      log_debug("[create] " + httpChannel.URI.spec + "\n");
      var new_data =  { http_channel : httpChannel,
                model : new XTraceFF_RequestModel(httpChannel.URI.spec) };
      ctx.requests.push(new_data);
      return new_data;
    }
  },

  drop_request : function( httpChannel ) {
    var ctx = this.context_for_subject(httpChannel);
    var idx = this.find_request(httpChannel);
    if(idx == -1) {
      return null;
    }
    log_debug("[destroy] " + httpChannel.URI.spec + "\n");
    return ctx.requests.splice(idx, 1)[0];
  },

  clear_all_requests : function( httpChannel ) {
    var ctx = this.context_for_subject(httpChannel);
    ctx.requests = [];
  }
};

XTraceFF_RequestModel = function(uri) {
  this.uri = uri;
};

XTraceFF_RequestModel.prototype = {
  log_activity : function( event_type, timestamp ) {
    if(this.hasOwnProperty("xtrmd")) {
      var options = {'Timestamp' : timestamp};
      if( event_type === 'sending-to' &&
```

```
          this.hasOwnProperty("branch_id") ) {
        options['BranchFork'] = this.branch_id;
      }

      this.xtrmd.log_event("HttpRequestObserver",
                           event_type,
                           options);

    }
  },

  fields : [ 'header-queued', 'header-sent', 'response-sent',
             'response-header-recvd', 'response-recvd', 'closed' ],
  to_html : function() {
    var msg = "";
    var fields = this.fields;
    for(var idx in fields) {
      var field = fields[idx];
      if(this.hasOwnProperty(field)) {
        msg = msg + "<p>" + field + ": " + (this[ field ])/1000 + "ms</p>";
      }
    }
    return msg;
  },

  log_events : function( xtrmd, optional ) {
    var options = { };
    for( var key in optional) {
      if(optional.hasOwnProperty(key)) {
        options[key] = optional[key];
      }
    }
    for(var idx in this.fields) {
      var field = this.fields[idx];
      if(this.hasOwnProperty(field)) {
        optional['Timestamp'] = this[field];
        xtrmd.log_event("HttpRequestObserver", field, optional);
      }
    }
  }
};

////////////////////////////////////////////////////////
// XTrace Metadata state
//
// This contains the state required to create & send X-Trace
// Metadata (xtr-md) along with methods to actually send it to
//  a listening server.
//
////////////////////////////////////////////////////////

XTraceFF_XtrContext = function(fromXtrmd) {
  // if fromXtrmd is set, split it apart and use that instead
  if(fromXtrmd === undefined) {
```

```
    this.taskId = this.generateHexForBytes(8);
    this.rootOpId = "0000000000000000";
  } else {
    this.taskId = fromXtrmd.slice(2, 18);
    this.rootOpId = fromXtrmd.slice(18, 34);
  }

  this.xtraceUrl = "http://luigi.usask.ca:7900/";
};

XTraceFF_XtrContext.prototype.get_xtrace_url = function() {
  var prefs = Components.classes[
    "@mozilla.org/preferences-service;1"]
    .getService(Components.interfaces.nsIPrefService);
  prefs = prefs.getBranch("extensions.mtwave1.");

  return prefs.getCharPref('mtwave_server');
};

XTraceFF_XtrContext.prototype.fork_event_chain = function() {
  var newCtx = new XTraceFF_XtrContext();
  newCtx.taskId = this.taskId;
  newCtx.rootOpId = this.rootOpId;
  return newCtx;
};

XTraceFF_XtrContext.prototype.generate_branch = function() {
  var branchId = this.generateHexForBytes(8);
//  var tracePlanId = this.generateHexForBytes(8);
//  var tracePlanId = "0011223344556677";
  var xtraceStr = '1D' + this.taskId + this.rootOpId;
  // branchId is also 8 bytes, + length + type = 10
  xtraceStr += '0A';
  xtraceStr += '80'; // branch option type
  xtraceStr += '08'; // length of field
  xtraceStr += branchId;
/*  xtraceStr += '81'; // trace plan type
  xtraceStr += '08'; // length of field
  xtraceStr += tracePlanId; */
  return {
    'branch_id' : branchId,
    'xtrmd_string' : xtraceStr
  };
};


XTraceFF_XtrContext.prototype.log_event = function(agent, label,
    optional_params, is_first) {
  // Log an xtrace event from agent with label.
  //
  // This will return an xtrace metadata string that you can use to
  // annotate a request (e.g. for attaching to an AJAX request)
  //
  // This will not advance the base opId, so that the causality chain
```

```
  // will  not accidentally imply a causal relationship when there is
  // not.  If you  want to imply a causal relationship, call push_opid
  // first, to generate a new causal relationship
  var newOpId;

  newOpId = this.generateHexForBytes(8);

  // 19 is hard-coded XTrace Protocol v1, 8-byte taskId, 8-byte opId
  var xtraceStr = '19' + this.taskId + newOpId;
  var event = { 'X-Trace' : xtraceStr,
                'Agent' : agent,
                'Label' : label,
                'Edge' : this.rootOpId,
                'Timestamp' : (new Date()).getTime() / 1000
              };

  if(optional_params !== undefined) {
    for(var key in optional_params) {
      if (optional_params.hasOwnProperty(key) &&
          (optional_params[key] !== null)) {
            event[key] = optional_params[key];
      }
    }
  }

  this.queue_event(event);
  this.rootOpId = newOpId;
  return xtraceStr;
};


XTraceFF_XtrContext.prototype.queue_event = function(event) {
  var nativeJSON = Components.classes["@mozilla.org/dom/json;1"]
    .createInstance(Components.interfaces.nsIJSON);
  var xtraceJsonEvent = nativeJSON.encode( event );

  // See https://developer.mozilla.org/En/
  //      Firefox_addons_developer_guide/
  //      Using_XPCOM-Implementing_advanced_processes
  // ...and https://developer.mozilla.org/En/
  //      Using_XMLHttpRequest
  // ...and https://developer.mozilla.org/en/XMLHttpRequest#send()
  log_debug( "event: " + xtraceJsonEvent + "\n");
  var req = Components.classes[
    "@mozilla.org/xmlextras/xmlhttprequest;1"]
    .createInstance(Components.interfaces.nsIXMLHttpRequest);

  var content = ('q=' + encodeURIComponent(xtraceJsonEvent))
    .replace(/%20/g, '+');

  var postData = Components.classes[
    '@mozilla.org/io/string-input-stream;1']
    .createInstance(Components.interfaces.nsIStringInputStream);
```

```
  postData.setData(content, content.length);

  // req.onprogress = onProgress;
  // req.onload = onLoad;
  // req.onerror = onError;
  req.open("POST", this.get_xtrace_url(), true);
  req.setRequestHeader(
    "Content-Type", "application/x-www-form-urlencoded");
  req.send(postData);
};

XTraceFF_XtrContext.prototype.generateHexForBytes = function(byteSize) {
  // generate an n-byte random string that would occupy
  // byteSize bytes, if it weren't hex encoded
  var digits = [];
  for(var i=0; i < byteSize; i++) {
    var intValue = Math.floor(Math.random()*256);
    var hexValue = intValue.toString(16);
    if(hexValue.length == 0) {
      hexValue = '00';
    } else if (hexValue.length == 1) {
      hexValue = '0' + hexValue;
    }
    digits.push(hexValue);
  }
  return digits.join('');
};
```

A.3 Ruby Tracing

```
tracer = TracePoint.new(:call, :return) do |tp|
  next if Thread.current[:md].nil?
  md = Thread.current[:md]
  next if md.inside
  md.inside = true
  case tp.event
  when :call
    md.indent_level += 1
    p ``#{md.tid} exec [#{tp.method_id}] (#{tp.path}:#{tp.lineno})''
  when :return
    if md.indent_level > 0 then
      md.indent_level -= 1
      p ``#{md.tid} end_exec''
    end
  end
  md.inside = false
end

module XTrace
  class Xtrmd
    def initialize(tid)
      self.tid = tid
      self.indent_level = 0
```

```
      self.inside = false
    end
    attr_accessor :indent_level, :tid, :inside
  end

  def self.bind_current_thread
    tid = sprintf(''%04x'', ((2**16)*Random.rand).to_i)
    md = Xtrmd.new(tid)
    p ''#{tid} binding''
    Thread.current[:md] = md
  end
end

class << Thread
  alias untraced_new new
  def new(*args, &block)
    untraced_new(*args) do
      XTrace::bind_current_thread
      block.call
    end
  end
end
```